CS 211: Introduction to Programming
Summer 2016 – Coding Style Guidelines

The rules below are not guidelines or recommendations, but strict rules. Your grade on your assignment depends on adherence to these rules.

In general, we will follow Linux Kernel Coding Style.

References on the Web:
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/CodingStyle?id=refs/heads/master

Write Short Functions

To the extent that it is feasible, functions should be kept small and focused. It is, however, recognized that long functions are sometimes appropriate, so no hard limit is placed on function length. If a function exceeds 40 lines or so, think about whether it can be broken up without harming the structure of the program.

Limit Variable Scope

The scope of local variables should be kept to a minimum. By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the beginning of the block they appear in.

Loop variables should be declared in the *for* statement itself (unless there is a compelling reason to do otherwise)

```
for (int i = 0; i < n; i++)
   doSomething(i);
```

Parting Thought…

BE CONSISTENT. If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their *if* clauses, you should too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding, so people can concentrate on what you're saying, rather than on how you're saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. **Avoid this.**

```
                    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 PI 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:

          if (condition) do_this;
            do_something_everytime;

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, and the above example is deliberately broken.
```

```
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:

          if (x is true) {
                    we do y
          }

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

          switch (action) {
          case KOBJ_ADD:
                    return "add";
          case KOBJ_REMOVE:
                    return "remove";
          case KOBJ_CHANGE:
                    return "change";
          default:
                    return NULL;
          }

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,
ie 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) {
                    ..
```

```
142             } else if (x > y) {
143                     ...
144             } else {
145                     ....
146             }
147
148     Rationale: K&R.
149
150     Also, note that this brace-placement also minimizes the number of empty
151     (or almost empty) lines, without any loss of readability.  Thus, as the
152     supply of new-lines on your screen is not a renewable resource (think
153     25-line terminal screens here), you have more empty lines to put
154     comments on.
155
156     Do not unnecessarily use braces where a single statement will do.
157
158             if (condition)
159                     action();
160
161     and
162
163             if (condition)
164                     do_this();
165             else
166                     do_that();
167
168     This does not apply if only one branch of a conditional statement is a single
169     statement; in the latter case use braces in both branches:
170
171             if (condition) {
172                     do_this();
173                     do_that();
174             } else {
175                     otherwise();
176             }
177
178                     3.1:  Spaces
179
180     Linux kernel style for use of spaces depends (mostly) on
181     function-versus-keyword usage.  Use a space after (most) keywords.  The
182     notable exceptions are sizeof, typeof, alignof, and __attribute__, which look
183     somewhat like functions (and are usually used with parentheses in Linux,
184     although they are not required in the language, as in: "sizeof info" after
185     "struct fileinfo info;" is declared).
186
187     So use a space after these keywords:
188
189             if, switch, case, for, do, while
190
191     but not with sizeof, typeof, alignof, or __attribute__.  E.g.,
192
193             s = sizeof(struct file);
194
195     Do not add spaces around (inside) parenthesized expressions.  This example is
196     *bad*:
197
198             s = sizeof( struct file );
199
200     When declaring pointer data or a function that returns a pointer type, the
201     preferred use of '*' is adjacent to the data name or function name and not
202     adjacent to the type name.  Examples:
203
204             char *linux_banner;
205             unsigned long long memparse(char *ptr, char **retptr);
206             char *match_strdup(substring_t *s);
207
208     Use one space around (on each side of) most binary and ternary operators,
209     such as any of these:
210
211             =  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
212
```

```
213 but no space after unary operators:
214
215           &  *  +  -  ~  !  sizeof  typeof  alignof   attribute    defined
216
217 no space before the postfix increment & decrement unary operators:
218
219           ++  --
220
221 no space after the prefix increment & decrement unary operators:
222
223           ++  --
224
225 and no space around the '.' and "->" structure member operators.
226
227 Do not leave trailing whitespace at the ends of lines.  Some editors with
228 "smart" indentation will insert whitespace at the beginning of new lines as
229 appropriate, so you can start typing the next line of code right away.
230 However, some such editors do not remove the whitespace if you end up not
231 putting a line of code there, such as if you leave a blank line.  As a result,
232 you end up with lines containing trailing whitespace.
233
234 Git will warn you about patches that introduce trailing whitespace, and can
235 optionally strip the trailing whitespace for you; however, if applying a series
236 of patches, this may make later patches in the series fail by changing their
237 context lines.
238
239
240                     Chapter 4: Naming
241
242 C is a Spartan language, and so should your naming be.  Unlike Modula-2
243 and Pascal programmers, C programmers do not use cute names like
244 ThisVariableIsATemporaryCounter.  A C programmer would call that
245 variable "tmp", which is much easier to write, and not the least more
246 difficult to understand.
247
248 HOWEVER, while mixed-case names are frowned upon, descriptive names for
249 global variables are a must.  To call a global function "foo" is a
250 shooting offense.
251
252 GLOBAL variables (to be used only if you _really_ need them) need to
253 have descriptive names, as do global functions.  If you have a function
254 that counts the number of active users, you should call that
255 "count_active_users()" or similar, you should _not_ call it "cntusr()".
256
257 Encoding the type of a function into the name (so-called Hungarian
258 notation) is brain damaged - the compiler knows the types anyway and can
259 check those, and it only confuses the programmer.  No wonder MicroSoft
260 makes buggy programs.
261
262 LOCAL variable names should be short, and to the point.  If you have
263 some random integer loop counter, it should probably be called "i".
264 Calling it "loop_counter" is non-productive, if there is no chance of it
265 being mis-understood.  Similarly, "tmp" can be just about any type of
266 variable that is used to hold a temporary value.
267
268 If you are afraid to mix up your local variable names, you have another
269 problem, which is called the function-growth-hormone-imbalance syndrome.
270 See chapter 6 (Functions).
271
272
273                     Chapter 5: Typedefs
274
275 Please don't use things like "vps_t".
276 It's a _mistake_ to use typedef for structures and pointers. When you see a
277
278           vps t a;
279
280 in the source, what does it mean?
281 In contrast, if it says
282
283           struct virtual_container *a;
```

```
284
285 you can actually tell what "a" is.
286
287 Lots of people think that typedefs "help readability". Not so. They are
288 useful only for:
289
290  (a) totally opaque objects (where the typedef is actively used to _hide_
291      what the object is).
292
293      Example: "pte_t" etc. opaque objects that you can only access using
294      the proper accessor functions.
295
296      NOTE! Opaqueness and "accessor functions" are not good in themselves.
297      The reason we have them for things like pte_t etc. is that there
298      really is absolutely _zero_ portably accessible information there.
299
300  (b) Clear integer types, where the abstraction _helps_ avoid confusion
301      whether it is "int" or "long".
302
303      u8/u16/u32 are perfectly fine typedefs, although they fit into
304      category (d) better than here.
305
306      NOTE! Again - there needs to be a _reason_ for this. If something is
307      "unsigned long", then there's no reason to do
308
309          typedef unsigned long myflags_t;
310
311      but if there is a clear reason for why it under certain circumstances
312      might be an "unsigned int" and under other configurations might be
313      "unsigned long", then by all means go ahead and use a typedef.
314
315  (c) when you use sparse to literally create a  new  type for
316      type-checking.
317
318  (d) New types which are identical to standard C99 types, in certain
319      exceptional circumstances.
320
321      Although it would only take a short amount of time for the eyes and
322      brain to become accustomed to the standard types like 'uint32_t',
323      some people object to their use anyway.
324
325      Therefore, the Linux-specific 'u8/u16/u32/u64' types and their
326      signed equivalents which are identical to standard types are
327      permitted -- although they are not mandatory in new code of your
328      own.
329
330      When editing existing code which already uses one or the other set
331      of types, you should conform to the existing choices in that code.
332
333  (e) Types safe for use in userspace.
334
335      In certain structures which are visible to userspace, we cannot
336      require C99 types and cannot use the 'u32' form above. Thus, we
337      use __u32 and similar types in all structures which are shared
338      with userspace.
339
340 Maybe there are other cases too, but the rule should basically be to NEVER
341 EVER use a typedef unless you can clearly match one of those rules.
342
343 In general, a pointer, or a struct that has elements that can reasonably
344 be directly accessed should _never_ be a typedef.
345
346
347                     Chapter 6: Functions
348
349 Functions should be short and sweet, and do just one thing.  They should
350 fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24,
351 as we all know), and do one thing and do that well.
352
353 The maximum length of a function is inversely proportional to the
354 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.  If the function is
exported, the EXPORT* macro for it should follow immediately after the closing
function brace line.  E.g.:

          int system_is_up(void)
          {
                    return system_state == SYSTEM_RUNNING;
          }
          EXPORT_SYMBOL(system_is_up);

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 7: Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is
used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple
locations and some common work such as cleanup has to be done.  If there is no
cleanup needed then just return directly.

Choose label names which say what the goto does or why the goto exists.  An
example of a good name could be "out_buffer:" if the goto frees "buffer".  Avoid
using GW-BASIC names like "err1:" and "err2:".  Also don't name them after the
goto location like "err_kmalloc_failed:"

The rationale for using gotos is:

- unconditional statements are easier to understand and follow
- nesting is reduced
- errors by not updating individual exit points when making
    modifications are prevented
- saves the compiler work to optimize redundant code away ;)

          int fun(int a)
          {
                    int result = 0;
                    char *buffer;

                    buffer = kmalloc(SIZE, GFP_KERNEL);
                    if (!buffer)
                              return -ENOMEM;

                    if (condition1) {
                              while (loop1) {
                                        ...
                              }
                              result = 1;
                              goto out_buffer;
```

```
                        }
                        ...
                out buffer:
                        kfree(buffer);
                        return result;
                }
```

A common type of bug to be aware of is "one err bugs" which look like this:

```
                err:
                        kfree(foo->bar);
                        kfree(foo);
                        return ret;
```

The bug in this code is that on some exit paths "foo" is NULL. Normally the fix for this is to split it up into two error labels "err bar:" and "err foo:".

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.

When commenting the kernel API functions, please use the kernel-doc format. See the files Documentation/kernel-doc-nano-HOWTO.txt and scripts/kernel-doc for details.

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

The preferred style for long (multi-line) comments is:

```
                /*
                 * This is the preferred style for multi-line
                 * comments in the Linux kernel source code.
                 * Please use it consistently.
                 *
                 * Description:  A column of asterisks on the left side,
                 * with beginning and ending almost-blank lines.
                 */
```

For files in net/ and drivers/net/ the preferred style for long (multi-line) comments is a little different.

```
                /* The preferred comment style for files in net/ and drivers/net
                 * looks like this.
                 *
                 * It is nearly the same as the generally preferred comment style,
                 * but there is no initial almost-blank line.
                 */
```

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.

Chapter 9: You've made a mess of it

That's OK, we all do. You've probably been told by your long-time Unix

```
user helper that "GNU emacs" automatically formats the C sources for
you, and you've noticed that yes, it does do that, but the defaults it
uses are less than desirable (in fact, they are worse than random
typing - an infinite number of monkeys typing into GNU emacs would never
make a good program).

So, you can either get rid of GNU emacs, or change it to use saner
values.  To do the latter, you can stick the following in your .emacs file:

(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
         (column (c-langelem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset)))

(add-hook 'c-mode-common-hook
          (lambda ()
            ;; Add kernel style
            (c-add-style
             "linux-tabs-only"
             '("linux" (c-offsets-alist
                        (arglist-cont-nonempty
                         c-lineup-gcc-asm-reg
                         c-lineup-arglist-tabs-only))))))

(add-hook 'c-mode-hook
          (lambda ()
            (let ((filename (buffer-file-name)))
              ;; Enable kernel mode for the appropriate files
              (when (and filename
                         (string-match (expand-file-name "~/src/linux-trees")
                                       filename))
                (setq indent-tabs-mode t)
                (setq show-trailing-whitespace t)
                (c-set-style "linux-tabs-only")))))

This will make emacs go better with the kernel coding style for C
files below ~/src/linux-trees.

But even if you fail in getting emacs to do sane formatting, not
everything is lost: use "indent".

Now, again, GNU indent has the same brain-dead settings that GNU emacs
has, which is why you need to give it a few command line options.
However, that's not too bad, because even the makers of GNU indent
recognize the authority of K&R (the GNU people aren't evil, they are
just severely misguided in this matter), so you just give indent the
options "-kr -i8" (stands for "K&R, 8 character indents"), or use
"scripts/Lindent", which indents in the latest style.

"indent" has a lot of options, and especially when it comes to comment
re-formatting you may want to take a look at the man page.  But
remember: "indent" is not a fix for bad programming.


                    Chapter 10: Kconfig configuration files

For all of the Kconfig* configuration files throughout the source tree,
the indentation is somewhat different.  Lines under a "config" definition
are indented with one tab, while help text is indented an additional two
spaces.  Example:

config AUDIT
          bool "Auditing support"
          depends on NET
          help
            Enable auditing infrastructure that can be used with another
            kernel subsystem, such as SELinux (which requires this for
```

```
            logging of avc messages output).  Does not do system-call
            auditing without CONFIG_AUDITSYSCALL.

Seriously dangerous features (such as write support for certain
filesystems) should advertise this prominently in their prompt string:

config ADFS_FS_RW
          bool "ADFS write support (DANGEROUS)"
          depends on ADFS_FS
          ...

For full documentation on the configuration files, see the file
Documentation/kbuild/kconfig-language.txt.


                    Chapter 11: Data structures

Data structures that have visibility outside the single-threaded
environment they are created and destroyed in should always have
reference counts.  In the kernel, garbage collection doesn't exist (and
outside the kernel garbage collection is slow and inefficient), which
means that you absolutely _have_ to reference count all your uses.

Reference counting means that you can avoid locking, and allows multiple
users to have access to the data structure in parallel - and not having
to worry about the structure suddenly going away from under them just
because they slept or did something else for a while.

Note that locking is _not_ a replacement for reference counting.
Locking is used to keep data structures coherent, while reference
counting is a memory management technique.  Usually both are needed, and
they are not to be confused with each other.

Many data structures can indeed have two levels of reference counting,
when there are users of different "classes".  The subclass count counts
the number of subclass users, and decrements the global count just once
when the subclass count goes to zero.

Examples of this kind of "multi-level-reference-counting" can be found in
memory management ("struct mm_struct": mm_users and mm_count), and in
filesystem code ("struct super_block": s_count and s_active).

Remember: if another thread can find your data structure, and you don't
have a reference count on it, you almost certainly have a bug.


                    Chapter 12: Macros, Enums and RTL

Names of macros defining constants and labels in enums are capitalized.

          #define CONSTANT 0x12345

Enums are preferred when defining several related constants.

CAPITALIZED macro names are appreciated but macros resembling functions
may be named in lower case.

Generally, inline functions are preferable to macros resembling functions.

Macros with multiple statements should be enclosed in a do - while block:

          #define macrofun(a, b, c)                       \
                    do {                                          \
                              if (a == 5)                         \
                                        do_this(b, c);            \
                    } while (0)

Things to avoid when using macros:

1) macros that affect control flow:
```

```
639             #define FOO(x)                                                  \
640                     do {                                                    \
641                             if (blah(x) < 0)                \
642                                     return -EBUGGERED; \
643                     } while (0)
644
```

is a _very_ bad idea. It looks like a function call but exits the "calling" function; don't break the internal parsers of those who will read the code.

2) macros that depend on having a local variable with a magic name:

```
650             #define FOO(val) bar(index, val)
651
```

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.

3) macros with arguments that are used as l-values: FOO(x) = y; will bite you if somebody e.g. turns FOO into an inline function.

4) forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
662             #define CONSTANT 0x4000
663             #define CONSTEXP (CONSTANT | 3)
664
```

5) namespace collisions when defining local variables in macros resembling functions:

```
668 #define FOO(x)                                  \
669 ({                                              \
670             typeof(x) ret;                      \
671             ret = calc_ret(x);          \
672             (ret);                              \
673 })
674
```

ret is a common name for a local variable - foo ret is less likely to collide with an existing variable.

The cpp manual deals with macros exhaustively. The gcc internals manual also covers RTL which is used frequently with assembly language in the kernel.

Chapter 13: Printing kernel messages

Kernel developers like to be seen as literate. Do mind the spelling of kernel messages to make a good impression. Do not use crippled words like "dont"; use "do not" or "don't" instead. Make the messages concise, clear, and unambiguous.

Kernel messages do not have to be terminated with a period.

Printing numbers in parentheses (%d) adds no value and should be avoided.

There are a number of driver model diagnostic macros in <linux/device.h> which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level: dev_err(), dev_warn(), dev_info(), and so forth. For messages that aren't associated with a particular device, <linux/printk.h> defines pr_notice(), pr_info(), pr_warn(), pr_err(), etc.

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. However debug message printing is handled differently than printing other non-debug messages. While the other pr_XXX() functions print unconditionally, pr_debug() does not; it is compiled out by default, unless either DEBUG is defined or CONFIG_DYNAMIC_DEBUG is set. That is true for dev_dbg() also, and a related convention uses VERBOSE_DEBUG to add dev_vdbg() messages to the ones already enabled by DEBUG.

Many subsystems have Kconfig debug options to turn on -DDEBUG in the

corresponding Makefile; in other cases specific files #define DEBUG. And when a debug message should be unconditionally printed, such as if it is already inside a debug-related #ifdef section, printk(KERN DEBUG ...) can be used.

Chapter 14: Allocating memory

The kernel provides the following general purpose memory allocators: kmalloc(), kzalloc(), kmalloc_array(), kcalloc(), vmalloc(), and vzalloc(). Please refer to the API documentation for further information about them.

The preferred form for passing a size of a struct is the following:

```
p = kmalloc(sizeof(*p), ...);
```

The alternative form where struct name is spelled out hurts readability and introduces an opportunity for a bug when the pointer variable type is changed but the corresponding sizeof that is passed to a memory allocator is not.

Casting the return value which is a void pointer is redundant. The conversion from void pointer to any other pointer type is guaranteed by the C programming language.

The preferred form for allocating an array is the following:

```
p = kmalloc_array(n, sizeof(...), ...);
```

The preferred form for allocating a zeroed array is the following:

```
p = kcalloc(n, sizeof(...), ...);
```

Both forms check for overflow on the allocation size n * sizeof(...), and return NULL if that occurred.

Chapter 15: The inline disease

There appears to be a common misperception that gcc has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 milliseconds. There are a LOT of cpu cycles that can go into these 5 milliseconds.

A reasonable rule of thumb is to not put inline at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you *know* the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the kmalloc() inline function.

Often people argue that adding inline to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, gcc is capable of inlining these automatically without help, and the maintenance issue of removing the inline when a second user appears outweighs the potential value of the hint that tells gcc to do something it would have done anyway.

Chapter 16: Function return values and names

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error-code integer (-Exxx = failure, 0 = success) or a "succeeded" boolean (0 = failure, non-zero = success).

```
781
782  Mixing up these two sorts of representations is a fertile source of
783  difficult-to-find bugs.  If the C language included a strong distinction
784  between integers and booleans then the compiler would find these mistakes
785  for us... but it doesn't.  To help prevent such bugs, always follow this
786  convention:
787
788           If the name of a function is an action or an imperative command,
789           the function should return an error-code integer.  If the name
790           is a predicate, the function should return a "succeeded" boolean.
791
792  For example, "add work" is a command, and the add_work() function returns 0
793  for success or -EBUSY for failure.  In the same way, "PCI device present" is
794  a predicate, and the pci_dev_present() function returns 1 if it succeeds in
795  finding a matching device or 0 if it doesn't.
796
797  All EXPORTed functions must respect this convention, and so should all
798  public functions.  Private (static) functions need not, but it is
799  recommended that they do.
800
801  Functions whose return value is the actual result of a computation, rather
802  than an indication of whether the computation succeeded, are not subject to
803  this rule.  Generally they indicate failure by returning some out-of-range
804  result.  Typical examples would be functions that return pointers; they use
805  NULL or the ERR_PTR mechanism to report failure.
806
807
808                     Chapter 17:  Don't re-invent the kernel macros
809
810  The header file include/linux/kernel.h contains a number of macros that
811  you should use, rather than explicitly coding some variant of them yourself.
812  For example, if you need to calculate the length of an array, take advantage
813  of the macro
814
815           #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
816
817  Similarly, if you need to calculate the size of some structure member, use
818
819           #define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
820
821  There are also min() and max() macros that do strict type checking if you
822  need them.  Feel free to peruse that header file to see what else is already
823  defined that you shouldn't reproduce in your code.
824
825
826                     Chapter 18:  Editor modelines and other cruft
827
828  Some editors can interpret configuration information embedded in source files,
829  indicated with special markers.  For example, emacs interprets lines marked
830  like this:
831
832           -*- mode: c -*-
833
834  Or like this:
835
836           /*
837           Local Variables:
838           compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
839           End:
840           */
841
842  Vim interprets markers that look like this:
843
844           /* vim:set sw=8 noet */
845
846  Do not include any of these in source files.  People have their own personal
847  editor configurations, and your source files should not override them.  This
848  includes markers for indentation and mode configuration.  People may use their
849  own custom mode, or may have some other magic method for making indentation
850  work correctly.
851
```

```
852
853                     Chapter 19:  Inline assembly
854
855  In architecture-specific code, you may need to use inline assembly to interface
856  with CPU or platform functionality.  Don't hesitate to do so when necessary.
857  However, don't use inline assembly gratuitously when C can do the job.  You can
858  and should poke hardware from C when possible.
859
860  Consider writing simple helper functions that wrap common bits of inline
861  assembly, rather than repeatedly writing them with slight variations.  Remember
862  that inline assembly can use C parameters.
863
864  Large, non-trivial assembly functions should go in .S files, with corresponding
865  C prototypes defined in C header files.  The C prototypes for assembly
866  functions should use "asmlinkage".
867
868  You may need to mark your asm statement as volatile, to prevent GCC from
869  removing it if GCC doesn't notice any side effects.  You don't always need to
870  do so, though, and doing so unnecessarily can limit optimization.
871
872  When writing a single inline assembly statement containing multiple
873  instructions, put each instruction on a separate line in a separate quoted
874  string, and end each string except the last with \n\t to properly indent the
875  next instruction in the assembly output:
876
877            asm ("magic %reg1, #42\n\t"
878                 "more_magic %reg2, %reg3"
879                 : /* outputs */ : /* inputs */ : /* clobbers */);
880
881
882                      Chapter 20: Conditional Compilation
883
884  Wherever possible, don't use preprocessor conditionals (#if, #ifdef) in .c
885  files; doing so makes code harder to read and logic harder to follow.  Instead,
886  use such conditionals in a header file defining functions for use in those .c
887  files, providing no-op stub versions in the #else case, and then call those
888  functions unconditionally from .c files.  The compiler will avoid generating
889  any code for the stub calls, producing identical results, but the logic will
890  remain easy to follow.
891
892  Prefer to compile out entire functions, rather than portions of functions or
893  portions of expressions.  Rather than putting an ifdef in an expression, factor
894  out part or all of the expression into a separate helper function and apply the
895  conditional to that function.
896
897  If you have a function or variable which may potentially go unused in a
898  particular configuration, and the compiler would warn about its definition
899  going unused, mark the definition as __maybe_unused rather than wrapping it in
900  a preprocessor conditional.  (However, if a function or variable *always* goes
901  unused, delete it.)
902
903  Within code, where possible, use the IS_ENABLED macro to convert a Kconfig
904  symbol into a C boolean expression, and use it in a normal C conditional:
905
906            if (IS_ENABLED(CONFIG_SOMETHING)) {
907                     ...
908            }
909
910  The compiler will constant-fold the conditional away, and include or exclude
911  the block of code just as with an #ifdef, so this will not add any runtime
912  overhead.  However, this approach still allows the C compiler to see the code
913  inside the block, and check it for correctness (syntax, types, symbol
914  references, etc).  Thus, you still have to use an #ifdef if the code inside the
915  block references symbols that will not exist if the condition is not met.
916
917  At the end of any non-trivial #if or #ifdef block (more than a few lines),
918  place a comment after the #endif on the same line, noting the conditional
919  expression used.  For instance:
920
921            #ifdef CONFIG_SOMETHING
922            ...
```

```
923           #endif /* CONFIG_SOMETHING */
924
925
926                     Appendix I: References
927
928   The C Programming Language, Second Edition
929   by Brian W. Kernighan and Dennis M. Ritchie.
930   Prentice Hall, Inc., 1988.
931   ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).
932
933   The Practice of Programming
934   by Brian W. Kernighan and Rob Pike.
935   Addison-Wesley, Inc., 1999.
936   ISBN 0-201-61586-X.
937
938   GNU manuals - where in compliance with K&R and this text - for cpp, gcc,
939   gcc internals and indent, all available from http://www.gnu.org/manual/
940
941   WG14 is the international standardization working group for the programming
942   language C, URL: http://www.open-std.org/JTC1/SC22/WG14/
943
944   Kernel CodingStyle, by greg@kroah.com at OLS 2002:
945   http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/
946
```