Toward a better list iterator for the kernel

By **Jonathan Corbet**
March 10, 2022

Linked lists are conceptually straightforward; they tend to be taught toward the beginning of entry-level data-structures classes. It might thus be surprising that the kernel community is concerned about its longstanding linked-list implementation and is not only looking for ways to solve some problems, but has been struggling to find that solution. It now appears that some improvements might be at hand: after more than 30 years, the kernel developers may have found a better way to safely iterate through a linked list.

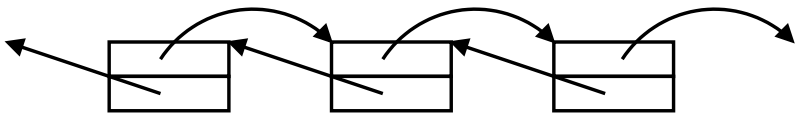
Kernel linked lists

C, of course, makes the creation of linked lists relatively easy. What it does not do, though, is help in the creation of *generic* linked lists that can contain any type of structure. By its nature, C lends itself to the creation of ad hoc linked lists in every situation where they are needed, resulting in boilerplate code and duplicated definitions. Every linked-list implementation must be reviewed for correctness. It would be far nicer to have a single implementation that was known to work so that kernel developers could more profitably use their time introducing bugs into code that is truly unique to their problem area.

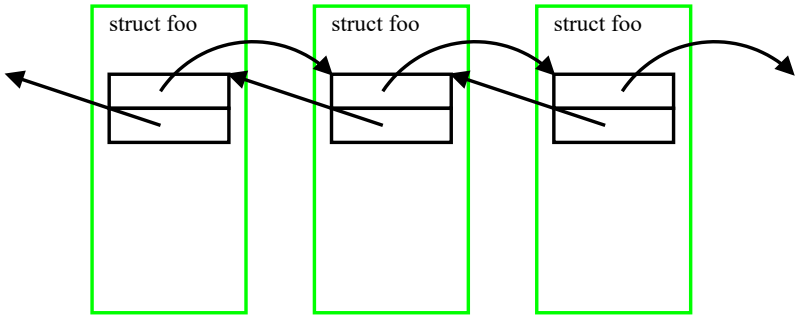
The kernel, naturally, has a few solutions for linked lists, but the most commonly used is `struct list_head`:

```
struct list_head {
    struct list_head *next, *prev;
};
```

This structure can be employed in the obvious way to create doubly linked lists; a portion of such a list might look like:

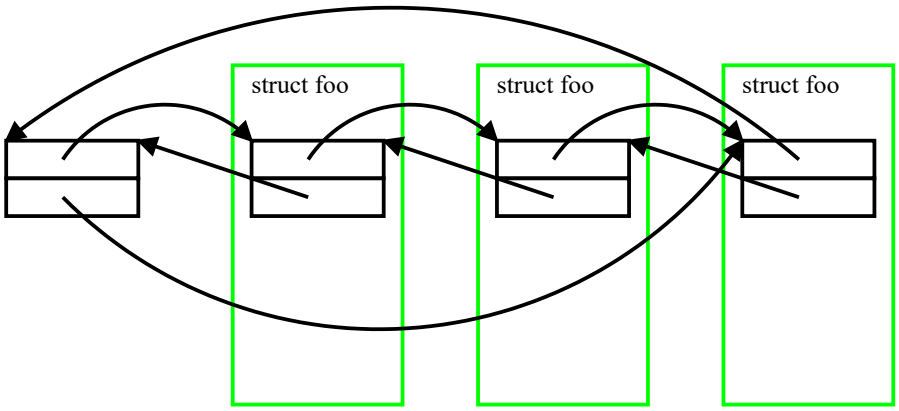


`struct list_head` can represent a linked list nicely, but has one significant disadvantage: it cannot hold any other information. Usually, this kind of data structure is needed to link some other data of interest; the list structure by itself isn't the point. C does not make it easy to create a linked list with an arbitrary payload, but it *is* easy to embed `struct list_head` inside the structure that the developer actually wants to organize into a list:



This is how linked lists are typically constructed in the kernel. Macros like `container_of()` can be used to turn a pointer to a `list_head` structure into a pointer to the containing structure. Code that works with linked lists will almost always use this macro (often indirectly) to gain access to the larger payload.

One final detail that is worthy of note is that the actual head of the list tends to be a `list_head` structure that is not embedded within the structure type of interest:



For a real-world example of how this infrastructure is used, consider `struct inode`, which represents a file within a filesystem. Inodes can be on a lot of lists simultaneously, so `struct inode` contains no less than five separate `list_head` structures; unfortunately, your editor's meager artistic skills are not up to the task of showing what the resulting data structure looks like. One of those `list_head` structures, `i_sb_list`, is used to associate the inode with the superblock of the filesystem it belongs to. The `list_head` structure that anchors this list is the `s_inodes` field of `struct super_block`. That is the one `list_head` structure in this particular list that is not embedded within an instance of `struct inode`.

Traversal of a linked list will typically begin at the anchor and follow the `next` pointers until the head is found again. One can, of course, open-code this traversal, but the kernel also provides a long list of functions and macros for this purpose. One of those is `list_for_each_entry()`, which will go through the entire list, providing a pointer to the containing structure at each node. Typical code using this macro looks like this:

```
struct inode *inode;

/* ... */
list_for_each_entry(inode, &sb->s_inodes, i_sb_list) {
    /* Process each inode here */
}
/* Should not use the iterator here */
```

Within the loop, the macro uses `container_of()` to point `inode` at the containing `inode` structure for each list entry. The problem is: what is the value of `inode` on exit from the loop? If code exited the loop with a `break` statement, `inode` will point to the element under consideration at that time. If, however, execution passes through the entire list, `inode` will be the result of using `container_of()` on the separate list head, which is not contained within an `inode` structure. That puts the kernel deeply into undefined-behavior territory and could lead to any of a number of bad things.

For this reason, the rule for macros like `list_for_each_entry()` is that the iterator variable should not be used outside of the loop. If a value needs to be accessed after the loop, it should be saved in a separate variable for that purpose. It's an implicit rule, though; nobody felt the need to actually note this restriction in the documentation for the macros themselves. Unsurprisingly, this rule is thus more of a guideline at best; the kernel is full of code that does, indeed, use the iterator variable after the loop.

The search for a safer iterator

When we last looked at this issue, Jakob Koschel had posted patches fixing some of these sites; he continued this project afterward. Linus Torvalds, however, thought that this approach was inadequate because it did nothing to prevent future problems from being introduced:

> So if we have the basic rule being "don't use the loop iterator after the loop has finished, because it can cause all kinds of subtle issues", then in _addition_ to fixing the existing code paths that have this issue, I really would want to (a) get a compiler warning for future cases and (b) make it not actually _work_ for future cases.
>
> Because otherwise it will just happen again.

Along the way, the developers came to the realization that moving to a newer version of the C standard might help, since it would allow the declaration of the iterator variable within the loop itself (thus making it invisible outside of the loop). Torvalds made an initial attempt at a solution that looked like this:

```
#define list_for_each_entry(pos, head, member)                          \
    for (typeof(pos) __iter = list_first_entry(head, typeof(*pos), member); \
         !list_entry_is_head(__iter, head, member) && (((pos)=__iter),1);   \
         __iter = list_next_entry(__iter, member))
```

This version of the macro still accepts the iterator variable as an argument, keeping the same prototype as before; this is important, since there are thousands of instances of this macro in the kernel source. But it declares a new variable to do the actual iteration, and only sets the passed-in iterator within the loop itself. Since the loop itself may never be executed (if the list is empty), the possibility exists that it will not set the iterator, so it could be uninitialized afterward.

This version was quickly followed by a second attempt, described as "a work of art":

```
#define list_for_each_entry(pos, head, member)                          \
    for (typeof(pos) pos = list_first_entry(head, typeof(*pos), member);    \
         !list_entry_is_head(pos, head, member);                        \
         pos = list_next_entry(pos, member))
```

Now the loop-scope iteration variable is declared with the same name as the outer variable, shadowing it. With this version, the iterator variable declared in the outer scope will never be used within the loop at all.

Torvalds's hope with both of these attempts was that this would cause the compiler to generate warnings if the (outer) iterator was used outside the loop, since it will no longer have been initialized by the loop itself. That did not work, though; there are places in the code that explicitly initialize the iterator now and, in any case, the "use of uninitialized variable" warning is disabled in the kernel due to excessive false positives.

James Bottomley suggested a different approach:

```
#define list_for_each_entry(pos, head, member)                          \
    for (pos = list_first_entry(head, typeof(*pos), member);            \
         !list_entry_is_head(pos, head, member) && ((pos = NULL) == NULL;   \
         pos = list_next_entry(pos, member))
```

This version would explicitly set the iterator variable to `NULL` on exit from the loop, causing any code that uses it to (presumably) fail. Torvalds pointed out the obvious problem with this attempt: it changes the semantics of a macro that is widely used throughout the kernel and would likely introduce bugs. It would also make life difficult for developers backporting patches to stable kernels that didn't have the newer semantics.

Yet another approach was proposed by Xiaomeng Tong:

```
#define list_for_each_entry_inside(pos, type, head, member)         \
    for (type * pos = list_first_entry(head, type, member);         \
         !list_entry_is_head(pos, head, member);                    \
         pos = list_next_entry(pos, member))
```

Tong's patch set created a new set of macros, with new names, with the idea that existing code could be converted over one usage at a time. There would be no externally declared iterator at all; instead, the name and type of the iterator are passed as arguments, and the iterator is declared within the scope of the loop itself. Torvalds, however, disliked this approach as well. Its use leads to long, difficult-to-read lines of code in almost every use and, he said, puts the pain in the wrong place: "We should strive for the *bad* cases to have to do extra work, and even there we should really strive for legibility".

A solution at last?

After having rejected various solutions, Torvalds went off to think about what a good solution might look like. Part of the problem, he concluded, is that the type of the containing structure is separate from the `list_head` structure, making the writing of iterator macros harder. If those two types could be joined somehow, things would be easier. Shortly thereafter, he came up with a solution that implements this idea. It starts with a new declaration macro:

```
#define list_traversal_head(type,name,target_member) \
   union { struct list_head name; type *name##_traversal_type; }
```

This macro would be used to declare the real head of the list — not the `list_head` entries contained within other structures. Specifically, it declares a variable of this new union type containing a `list_head` structure called `name`, and a pointer to the containing structure `type` called *`name`*`_traversal_type`. The pointer is never used as such; it is just a way of tying the type of the containing structure to the `list_head` variable.

Then, there is a new iterator:

```
#define list_traverse(pos, head, member) \
    for (typeof(*head##_traversal_type) pos = list_first_entry(head, typeof(*pos), member);\
        !list_entry_is_head(pos, head, member);     \
        pos = list_next_entry(pos, member))
```

Code can walk through a list by using `list_traverse()` instead of `list_for_each_entry()`. The iterator variable will be `pos`; it will only exist within the loop itself. The anchor of the list is passed as `head`, while `member` is the name of the `list_head` structure within the containing structure. The patch includes a couple of conversions to show what the usage would look like.

This, Torvalds thinks, is "the way forward". Making this change is probably a years-long project; there are over 15,000 uses of `list_for_each_entry()` (and variants) within the kernel. Each of those will eventually need to be changed, and the declaration of the list anchor must also change at the same time. So it is not a quick fix, but it could lead to a safer linked-list implementation in the kernel in the long run.

One might argue that all of this is self-inflicted pain caused by the continued use of C in the kernel. That may be true, but better alternatives are in somewhat short supply. For example, since the Rust language, for all of its merits, does not make life easy for anybody wanting to implement a linked list, a switch to that language would not automatically solve the problem. So kernel developers seem likely to have to get by with this kind of tricky infrastructure for some time yet.

Toward a better list iterator for the kernel

Posted Mar 10, 2022 16:06 UTC (Thu) by **pj** (subscriber, #4506) [Link]

I read part of that thread and what came to mind to me was that the devs were fighting the C preprocessor, and I was reminded of Zig trying to basically be (arguably) C without the cpp. Maybe once it's done Zig will manage to make it into the kernel - it's a better candidate than Rust, IMO, being much more C-like and having full C binary compatibility.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 20:04 UTC (Thu) by **pankyraghv** (subscriber, #153275) [Link]

I know many people in the kernel community hates cpp but I would recommend everyone to take a look at SerenityOS. They have done a fantastic job of rolling out an in-house standard library that actually makes coding in cpp enjoyable (mostly) while reaping the benefits of all the modern features that can catch bugs at compile time. IMO a subset of cpp with a good custom standard library would be a better fit to writing kernel drivers than use something like Rust which has a very different programming paradigm. I also feel something light like Zig might also be a good option.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 11, 2022 18:55 UTC (Fri) by **timon** (subscriber, #152974) [Link]

As I read this, you misunderstood the "cpp" part; the OP is talking about the C preprocessor and you are talking about C++.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 13, 2022 10:08 UTC (Sun) by **PengZheng** (subscriber, #108006) [Link]

Thanks for pointing me to SerenityOS. It seems that memory allocation failure will lead to crash:
https://github.com/SerenityOS/serenity/blob/master/AK/Vec...

I assume (it might be false) that C++ exception is not suitable for kernel development and some constrained embedded environment.
Are there any good guidelines for such C++ usages?

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 13, 2022 11:39 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

Basically, anywhere you are hardware-constrained (and an OS is a perfect example) you need to be very careful about knowing exactly what abstractions the compiler is using. C++ makes this rather tricky!

Once you're in user space, especially if you're in an online state (ie the computer is a lot faster than the person behind the keyboard), you no longer need to worry and can use the full power of C++. Just make sure you don't use the bits you don't understand :-)

Cheers,
Wol

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 14, 2022 2:26 UTC (Mon) by **PengZheng** (subscriber, #108006) [Link]

Unfortunately, code size will be a concern, even in user space. Nowadays, devices with 64MB flash storage does a lot of things, doubling code-size as mentioned in ianmcc's comment is terrifying.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 13, 2022 15:43 UTC (Sun) by **ianmcc** (subscriber, #88379) [Link]

It depends on how exceptions are implemented by the compiler. Exceptions require RTTI, because you need to be able to match the type of the thrown object with a corresponding catch (...) clause. When you have a try {...} block, the compiler needs to know how to unwind the stack. Most modern compilers use a 'zero overhead' model, meaning that the runtime cost (in CPU time) is basically zero for the no-exception case. ie, there is no penalty for having a try {...} block if no exception is actually thrown. The way they do this is with a lookup table from the instruction pointer to the exception handler for the current stack frame. This basically doubles the code size. Also the cost of throwing an exception is quite high, so you normally only want to do it for really exceptional events, not as an alternative to a return statement. But in some cases exceptions can be faster than return codes, i.e. since you need to check the return code every time, in a tight loop using exceptions might be faster.

More info: https://wg21.link/p1947

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 21:47 UTC (Thu) by **bartoc** (subscriber, #124262) [Link]

I'm concerned about how long zig tends to defer typechecking of code. Maybe it doesn't matter in the real world though.

I really love how zig does generics; you write type constructors as just .... normal functions that return a type!

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 22:47 UTC (Thu) by **roc** (subscriber, #30627) [Link]

The main point of Rust in the kernel is to get much-improved safety properties. Zig doesn't provide that.

Reply to this comment

My linked lists

Posted Mar 10, 2022 16:15 UTC (Thu) by **abatters** (★ **supporter** ★, #6932) [Link]

For non-circular linked lists in my userspace code, my approach is to define a dedicated macro for each specific list, defining the name of its head, tail, forward, and back pointers. Then that specific macro can be used in combination with one of ~25 generic doubly-linked list macros I have defined. Works well for having a single object in multiple lists, and having multiple lists in a single container. No iteration macro is necessary, since you have pointers to the real types (no container_of needed) and it terminates with NULL.

```
// Generic doubly-linked list macro (one of ~25)
#define DLIST_ADD_TAIL(head, tail, forw, back, item) \
   do { \
      if ((tail) == NULL) { \
```

```
         (head) = (item); \
      } else { \
         (item)->back = (tail); \
         (tail)->forw = (item); \
      } \
      (tail) = (item); \
   } while (0)

// Example linked list structs
struct item {
  struct item *iforw, *iback;
};
struct container {
  struct item *ihead, *itail;
};

// Example macro defining a specific list
#define CONTAINER_ITEM_LIST(container, func, args...) \
  func((container)->ihead, (container)->itail, iforw, iback , ## args)

void container_add_item_tail(struct container *container, struct *item)
{
   CONTAINER_ITEM_LIST(container, DLIST_ADD_TAIL, item);
}
```

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 17:08 UTC (Thu) by **jengelh** (subscriber, #33263) [Link]

>One might argue that all of this is self-inflicted pain caused by the continued use of C in the kernel. That may be true, but better alternatives are in somewhat short supply.

I disagree; alternatives are not in short supply, but they have tradeoffs that some people may be unwilling to go with. The Linux kernel linked list implementation has two properties:

- the linked list metadata (prev/next) pointers is not separated from the struct => layering violation
- an object can be part of multiple linked list => who's the owner responsible for cleanup?
(- the iterator interface of list_head is just a consequence of the two)

And these are approached with

- inode is, in terms of responsibility, strictly a child of the list and/or its internal metadata
- only one owner is allowed so there is no doubt who has to clean up

Using C++ as a concrete example now, one would arrive at:

1. Use std::list<std::shared_ptr<inode>>. Safe from leaks, safe from dangling pointers, but it needs refcounting to do the job, and some people may not like the performance characteristics of that refcounting.
2. Use std::list<inode> for the main list, and std::list<inode *> for the secondaries. Cleanup is still guaranteed, but you traded refcounting for the potential danger of dangling pointers.
(3. Keep on allocating your inodes and lists manually like before... as C++ is almost source-compatible with C.)

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 17:52 UTC (Thu) by **EnigmaticSeraph** (subscriber, #50582) [Link]

The kernel generally tolerates no performance hits, esp. with such a widely-used structure. And while there exists e.g.:
https://www.boost.org/doc/libs/1_78_0/doc/html/intrusive/...
, which can be configured to be as would've been at the C level, but generic. However, kernel devs tend to be wary of C++ in the kernel, for not-unfounded reasons.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 21:55 UTC (Thu) by **bartoc** (subscriber, #124262) [Link]

Note that what the list iterator does is pretty far from how C++ iterators work. This style of iterator is "I write the loop, and I'll insert your body with the right item all set up", where C++ is "you write the loop, and ask me for each item"

There's nothing really preventing you from doing the C++ approach in C, although you need to be careful about inlining, you just write an "iterator init from structure" "iterator next" and "is iterator done" function and call them in the appropriate places in a for loop. However I think the more intrusive way of doing things is quite a lot clearer when reading the iterator implementation, esp for more complicated iterators.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 12, 2022 2:59 UTC (Sat) by **droundy** (subscriber, #4559) [Link]

If C had the ability rust has to create a new variable shadowing the name of a former one, the macro could just create a new variable of a the same name but a different type to make any further use cause a compile error.

But of course if this weren't C we wouldn't have this problem.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 12, 2022 17:04 UTC (Sat) by **alonz** (subscriber, #815) [Link]

Actually in C11 it **is** possible to create a shadowing variable. However, this usually triggers a compiler warning - since such shadowing is rather confusing to the readers, and is usually considered bad practice (and a harbinger of confusing bugs).

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 14, 2022 16:43 UTC (Mon) by **laarmen** (subscriber, #63948) [Link]

In Rust, variable name re-use is actually rather idiomatic. I used to be skeptical about it, as my experience also was of confusing bugs, or at least code. However, I've embraced this idiom in Rust for purely pragmatic reasons:

In C, your 'char* str' variable will be the same type before and after your null check, but in Rust those are two separate types. A given "content" can undergo multiple type changes in as many lines as the

assumptions are checked, it'd be really tedious to have to come up with new names for essentially the same content.

You can find an example of shadowing in the Rust Book in the second chapter! https://doc.rust-lang.org/book/ch02-00-guessing-game-tuto... (the 'guess' name is reused as we go from the raw input string to the actual integer)

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 14, 2022 19:51 UTC (Mon) by **jem** (subscriber, #24231) [Link]

The difference between C and Rust is that Rust allows you reuse a variable name in the same block, which is an error i C.

The scope of a Rust variable variable ends either at the end of the block, or when a new variable with the same name is declared. You can use the old variable when computing the initial value of the new variable, i.e. you can initialize a variable with the value of the variable "itself".

This can be used to change a variable from being mutable to immutable when there is no need to change the value after some point in the program. (Technically there are two separate variables with the same name.)

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 13, 2022 16:39 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

> >One might argue that all of this is self-inflicted pain caused by the continued use of C in the kernel. That may be true, but better alternatives are in somewhat short supply.

> I disagree; alternatives are not in short supply, but they have tradeoffs that some people may be unwilling to go with. The Linux kernel linked list implementation has two properties:

Please note you are disagreeing with a straw man. Jon didn't write what you imply he did - you've elided the very important word *better*.

Cheers,
Wol

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 22:13 UTC (Thu) by **ppisa** (subscriber, #67307) [Link]

I have need for poor STL relative in C year ago and uLUt library was a result. Because it has been intended even for use in kernel at the times of RTLinux I have decide to be compatible/reuse kernel list but provide type-safe interfaces for each relation

https://sourceforge.net/p/ulan/ulut/ci/master/tree/ulut/u...

This way you never use container_of directly and risk to use incorrect offset between list node location and start of containing structure is suppressed. For AVL, hashes and sorted arrays I have used other structures than

kernel. The library provides even iterators which works over all of these basic types.

But macros which expands to the static inline functions are really huge. But reasonable compilers generate code better for these variants with CUSTom prefix than when generic functions with void pointers are used.

The library is base in many project, our RS485 multimaster project http://ulan.sourceforge.net/ , CAN and CANopen http://ortcan.sourceforge.net/ more instruments using inhouse our SmallToolKit graphic library https://gitlab.com/pikron/sw-base/suitk/-/wikis/home which has been used even is safety sensitive medical devices etc.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 22:50 UTC (Thu) by **roc** (subscriber, #30627) [Link]

> the "use of uninitialized variable" warning is disabled in the kernel due to excessive false positives.

Wow, this is terrible. I had no idea.

Given there are hardening patches to force zeroing of local variables, why not just initialize them in the source as necessary to eliminate the existing warnings and turn that warning on?

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 10, 2022 23:53 UTC (Thu) by **NYKevin** (subscriber, #129325) [Link]

If the compiler wrongly thinks that a variable is used before initialization, then it will probably fail to optimize away the dead store resulting from eager initialization. Whether that dead store is worth worrying about is another matter, of course, and probably depends on how hot a given codepath is.

Reply to this comment

Initialization

Posted Mar 11, 2022 0:29 UTC (Fri) by **tialaramex** (subscriber, #21167) [Link]

There was a CPP Con talk some years back about exactly this work by Microsoft. Basically let us teach the compiler to zero initialize everything, then run tests to see how awful the results are, teach the compiler to optimize away these dead stores and/or perform zero initialization that it more easily recognises as dead stores, iterate. IIRC The first few loops are very, very bad, nothing you could possibly ship, but as they learn what they're doing by the end they can more or less do this over the bulk of the system and get the same performance as before but with more confidence there aren't scary initialisation bugs. Obviously they were in C++ and their own C++ compiler, where Linux would want C and GCC, but it's not so different otherwise. If people can't find it themselves I can search Youtube maybe.

Reply to this comment

Initialization

Posted Mar 11, 2022 1:25 UTC (Fri) by **bartoc** (subscriber, #124262) [Link]

MSVC's behavior is a bit of a problem because there's no great opt-out, and it doesn't really do the optimizations in debug mode, meaning if you ever use stack arrays it's really easy to have awful debug

mode performance. It's one of the bigger "MSVC debug perf sucks" things around (along with like, not having a way to write inline functions that are always inlined, even in debug).

If there was an opt-out, and if MSVC had a saner "debug mode" (really these optimizations should be turned on in debug mode imo) then it would be a great behavior.

Reply to this comment

Initialization

Posted Mar 11, 2022 1:29 UTC (Fri) by **bartoc** (subscriber, #124262) [Link]

Note, I'm talking more about /RTCs rather than "InitAll", which might work better.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 11, 2022 1:24 UTC (Fri) by **roc** (subscriber, #30627) [Link]

My point is that a lot of those "dead stores" are already being performed by kernel hardening. https://github.com/torvalds/linux/blob/186d32bbf034417b40...

So on one hand we've got developers saying "adding dead stores hurts performance, don't do it and keep that essential warning disabled", and on the other hand developers are adding dead stores all over the place for the sake of security.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 11, 2022 11:20 UTC (Fri) by **geert** (subscriber, #98403) [Link]

Mindlessly adding preinitializations to uninitialized variables instead of fixing the real problems prevents the compiler from flagging newly introduced uninitialized users of the same variable later.

BTW, I used to fix all newly introduced non-false positives reported by gcc-4.1.2, which was rather strict (compared to later gcc versions), and caused lots of false positives. But every kernel release, I did catch a handful of real bugs. Later, gcc improved, Arnd started to build the kernel with bleeding edge gcc versions catching more bugs before I found them, and support for gcc-4.1.2 was dropped. But I believe there still are areas for improvements...

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 12, 2022 17:48 UTC (Sat) by **moorray** (subscriber, #54145) [Link]

I suggested exactly what Linus "came up with" a week later - solution #2 here: https://lore.kernel.org/all/20220228223228.24cf3fd4@kicin...

I wonder how these things happen...

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 12, 2022 19:28 UTC (Sat) by **Wol** (subscriber, #4433) [Link]

The same way all these things happen.

Alexander Graham Bell is credited with the invention of the telephone, but he just happened to beat his competitor to the patent office. I think the same is true of the Wright Brothers (the first successful aircraft was built in 1896 in England - it just didn't actually fly til about 10 years later ... :-)

These things are probably obvious to any competent mind that decides it wants to tackle the problem.

Cheers,
Wol

Reply to this comment

Inventors and heroes

Posted Mar 13, 2022 20:13 UTC (Sun) by **marcH** (subscriber, #57642) [Link]

> I think the same is true of the Wright Brothers (the first successful aircraft was built in 1896 in England - it just didn't actually fly til about 10 years later ... :-)

If you're French you know that the "first" flight was performed by either Clément Ader or Louis Blériot, if you're Brazilian then you know that it was Alberto Santos-Dumont, if you're German then maybe it's Gustav Weißkopf / Whitehead? Etc.

Everyone is correct as long as you tweak the definition of "first flight".

> These things are probably obvious to any competent mind that decides it wants to tackle the problem.

I wouldn't say "obvious" but Isaac Newton (one of the very few who was _really_ way ahead of his time) put it best: "If I have seen further it is by standing on the shoulders of Giants." This sentence was apparently inspired by older, similar sentences in an interesting, "recursive" twist.

Humanity is a very social species who can't achieve anything alone yet we love celebrating heroes. Inventors' contribution to progress is nothing compared to the Excel spreadsheets of the Department of Education but heroes make much better stories and movies and are a little bit more... inspirational! Heroes are especially important in American culture where the gaps are the widest; dispensing hope is so much cheaper. Panem, circenses and... "spes"?

> but he just happened to beat his competitor to the patent office.

Exactly: whatever you do, make sure you get some lawyers and businessmen on your side. History shows you can't make a difference without their help.

Reply to this comment

Inventors and heroes

Posted Mar 13, 2022 20:32 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

> I wouldn't say "obvious" but Isaac Newton (one of the very few who was _really_ way ahead of his time) put it best: "If I have seen further it is by standing on the shoulders of Giants." This sentence was apparently inspired by older, similar sentences in an interesting, "recursive" twist.

Was he? I think he was probably one of the founding members of the Royal Society, and many of them were equally "ahead of their time" - or was it we just have no real record of their predecessors? (Because before the Royal Society there *were* no records? Just like Shakespeare is "the greatest English playwright", not because he was necessarily any good, but also because he was the *first* major English playwright, writing at the time "modern English" was born.)

And while Newton wrote "In Principia Mathematica", I think it was primarily written to beat his contemporaries into the history books. Certainly it's disputed whether he was the "inventor" of Calculus, and the same is probably true of most of his other advances.

From what I can make out, the rivalry was intense, and Newton just won the publicity war.

Cheers,
Wol

Reply to this comment

Inventors and heroes

Posted Mar 13, 2022 22:16 UTC (Sun) by **NYKevin** (subscriber, #129325) [Link]

The current consensus is that Newton and Leibniz invented calculus at more or less the same time, independently of each other, but (in the English-speaking world at least) Newton had better publicity.

Interestingly, they also had very different ideas of what calculus was good for. Newton wanted to use it to model the physical world, while Leibniz seemed to think it had more metaphysical/philosophical significance. Arguably, they were both right, because the notion of Taylor series and analytic continuation have greatly improved our intuition and understanding of What Functions Are... and that has in turn been used to bring complex analysis into the world of physics and engineering (e.g. in the form of modern Fourier analysis). It's all connected in the end.

Reply to this comment

Inventors and heroes

Posted Mar 14, 2022 15:53 UTC (Mon) by **ballombe** (subscriber, #9523) [Link]

Being French, I will argue that Fermat invented calculus before them!
Really, ideas only exist when they are shared, and for that you need to be at least two.

While Hilbert gave his theorems numbers (90, 92 are the most well known)
Poincarré gave them the names of people whose work inspired him.

Reply to this comment

Inventors and heroes

Posted Mar 13, 2022 22:33 UTC (Sun) by **roc** (subscriber, #30627) [Link]

Shakespeare wasn't just first, he actually was incredibly good.

Reply to this comment

Inventors and heroes

Posted Mar 14, 2022 0:13 UTC (Mon) by **marcH** (subscriber, #57642) [Link]

> > Isaac Newton (one of the very few who was _really_ way ahead of his time)

> Certainly it's disputed whether he was the "inventor" of Calculus, and the same is probably true of most of his other advances.

OK, but:

> > heroes make much better stories and movies

So here it is: https://youtu.be/gMlf1ELvRzc "Then Newton came along and changed the game"

;-)

There is a genuine mathematician in awe of Newton in this video, so I think he was pretty far ahead. I have no idea whether he was ahead _alone_.

> or was it we just have no real record of their predecessors? (Because before the Royal Society there *were* no records?)

I'm not an expert but I'm pretty sure we have plenty of scientific records much, much older than Newton. Newton lived more than 200 years after Gutenberg so I really doubt that was still a problem at a time.

Reply to this comment

Inventors and heroes

Posted Mar 17, 2022 8:56 UTC (Thu) by **mathstuf** (subscriber, #69389) [Link]

The "shoulders of giants" quote was also meant as a slight to Leibnitz who was not tall.

Reply to this comment

Inventors and heroes

Posted Mar 17, 2022 13:05 UTC (Thu) by **excors** (subscriber, #95769) [Link]

That was Hooke, not Leibniz. (Newton seems to have been rather antagonistic and had several bitter rivalries. And as Master of the Mint he also had dozens of counterfeiters prosecuted and hanged, so those people probably weren't too happy with him either. But (to bring this marginally back on-topic) as Linus Torvalds said when asked about his heroes, "even though [Newton] was apparently not a very nice person, he was certainly one of those people who changed the world in many different ways" (https://archive.computerhistory.org/resources/access/text...))

Reply to this comment

Inventors and heroes

Posted Mar 20, 2022 10:24 UTC (Sun) by **ghane** (subscriber, #1805) [Link]

> I wouldn't say "obvious" but Isaac Newton (one of the very few who was _really_ way ahead of his time) put it best: "If I have seen further it is by standing on the shoulders of Giants."

The way I understand it, it was a dig at his arch-rival (in the 1670s), Hooke, who was short (and may have had a twisted back).

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 13, 2022 20:23 UTC (Sun) by **marcH** (subscriber, #57642) [Link]

https://www.ted.com/talks/linus_torvalds_the_mind_behind_...

> Well, so this is kind of cliché in technology, the whole Tesla versus Edison, where Tesla is seen as the visionary scientist and crazy idea man. And people love Tesla. I mean, there are people who name their companies after him.

> The other person there is Edison, who is actually often vilified for being kind of pedestrian and is -- I mean, his most famous quote is, "Genius is one percent inspiration and 99 percent perspiration." And I'm in the Edison camp, even if people don't always like him. Because if you actually compare the two, Tesla has kind of this mind grab these days, but who actually changed the world? Edison may not have been a nice person, he did a lot of things -- he was maybe not so intellectual, not so visionary. But I think I'm more of an Edison than a Tesla.

PS: funny enough Elon Musk was there early but not at the very beginning of Tesla. There's a great interview of the Tesla founders where they (among others) acknowledge everything that came before Tesla.

Reply to this comment

Independent invention

Posted Mar 19, 2022 20:54 UTC (Sat) by **giraffedata** (guest, #1954) [Link]

> Alexander Graham Bell is credited with the invention of the telephone, but he just happened to beat his competitor to the patent office. I think the same is true of the Wright Brothers (the first successful aircraft was built in 1896 in England - it just didn't actually fly til about 10 years later ... :-)

Being the first to fly, first to file and the first to publicize count for a lot. I think it's vanishingly rare for someone to be so smart he comes up with an idea that no one else has; turning such an idea into an invention is more worthy of credit than having the idea.

I've read that many things were invented in China before they were invented in Europe (in days before the two were in communication), but it's because of the European inventor that I have them today, so I tend to celebrate the European inventor.

Reply to this comment

I find myself rejoicing...

Posted Mar 18, 2022 1:43 UTC (Fri) by **smitty_one_each** (subscriber, #28989) [Link]

...that even the Really Smart Folks have to bang their heads as much as I do trying to get things working.

Reply to this comment

Toward a better list iterator for the kernel

Posted Mar 26, 2022 7:09 UTC (Sat) by **mcortese** (guest, #52099) [Link]

> Am I the only one who thinks that depending on the names of the variables is extremely fragile? There are so many corner cases where this hack will fail that I wonder if developers can really be convinced to begin using it.

Reply to this comment