The tip tree handbook

What is the tip tree?

The tip tree is a collection of several subsystems and areas of development. The tip tree is both a direct development tree and a aggregation tree for several sub-maintainer trees. The tip tree gitweb URL is: https://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git

The tip tree contains the following subsystems:

- **x86 architecture**

  The x86 architecture development takes place in the tip tree except for the x86 KVM and XEN specific parts which are maintained in the corresponding subsystems and routed directly to mainline from there. It's still good practice to Cc the x86 maintainers on x86-specific KVM and XEN patches.

  Some x86 subsystems have their own maintainers in addition to the overall x86 maintainers. Please Cc the overall x86 maintainers on patches touching files in arch/x86 even when they are not called out by the MAINTAINER file.

  Note, that `x86@kernel.org` is not a mailing list. It is merely a mail alias which distributes mails to the x86 top-level maintainer team. Please always Cc the Linux Kernel mailing list (LKML) `linux-kernel@vger.kernel.org`, otherwise your mail ends up only in the private inboxes of the maintainers.

- **Scheduler**

  Scheduler development takes place in the -tip tree, in the sched/core branch - with occasional sub-topic trees for work-in-progress patch-sets.

- **Locking and atomics**

  Locking development (including atomics and other synchronization primitives that are connected to locking) takes place in the -tip tree, in the locking/core branch - with occasional sub-topic trees for work-in-progress patch-sets.

- **Generic interrupt subsystem and interrupt chip drivers**:
  - interrupt core development happens in the irq/core branch
  - interrupt chip driver development also happens in the irq/core branch, but the patches are usually applied in a separate maintainer tree and then aggregated into irq/core

- **Time, timers, timekeeping, NOHZ and related chip drivers**:
  - timekeeping, clocksource core, NTP and alarmtimer development happens in the timers/core branch, but patches are usually applied in a separate maintainer tree and then aggregated into timers/core
  - clocksource/event driver development happens in the timers/core branch, but patches are mostly applied in a separate maintainer tree and then aggregated into timers/core

- **Performance counters core, architecture support and tooling**:
  - perf core and architecture support development happens in the perf/core branch
  - perf tooling development happens in the perf tools maintainer tree and is aggregated into the tip tree.

- **CPU hotplug core**

- **RAS core**

  Mostly x86-specific RAS patches are collected in the tip ras/core branch.

- **EFI core**

  EFI development in the efi git tree. The collected patches are aggregated in the tip efi/core branch.

- **RCU**

  RCU development happens in the linux-rcu tree. The resulting changes are aggregated into the tip core/rcu branch.

- **Various core code components**:
  - debugobjects
  - objtool
  - random bits and pieces

Patch submission notes

Selecting the tree/branch

In general, development against the head of the tip tree master branch is fine, but for the subsystems which are maintained separately, have their own git tree and are only aggregated into the tip tree, development should take place against the relevant subsystem tree or

branch.

Bug fixes which target mainline should always be applicable against the mainline kernel tree. Potential conflicts against changes which are already queued in the tip tree are handled by the maintainers.

Patch subject

The tip tree preferred format for patch subject prefixes is 'subsys/component:', e.g. 'x86/apic:', 'x86/mm/fault:', 'sched/fair:', 'genirq/core:'. Please do not use file names or complete file paths as prefix. 'git log path/to/file' should give you a reasonable hint in most cases.

The condensed patch description in the subject line should start with a uppercase letter and should be written in imperative tone.

Changelog

The general rules about changelogs in the process documentation, see :ref:`Documentation/process/ <submittingpatches>`, apply.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master)(Documentation)(process)maintainer-tip.rst, line 131); *backlink***
>
> Unknown interpreted text role "ref".

The tip tree maintainers set value on following these rules, especially on the request to write changelogs in imperative mood and not impersonating code or the execution of it. This is not just a whim of the maintainers. Changelogs written in abstract words are more precise and tend to be less confusing than those written in the form of novels.

It's also useful to structure the changelog into several paragraphs and not lump everything together into a single one. A good structure is to explain the context, the problem and the solution in separate paragraphs and this order.

Examples for illustration:

Example 1:

```
x86/intel_rdt/mbm: Fix MBM overflow handler during hot cpu

When a CPU is dying, we cancel the worker and schedule a new worker on a
different CPU on the same domain. But if the timer is already about to
expire (say 0.99s) then we essentially double the interval.

We modify the hot cpu handling to cancel the delayed work on the dying
cpu and run the worker immediately on a different cpu in same domain. We
donot flush the worker because the MBM overflow worker reschedules the
worker on same CPU and scans the domain->cpu_mask to get the domain
pointer.
```

Improved version:

```
x86/intel_rdt/mbm: Fix MBM overflow handler during CPU hotplug

When a CPU is dying, the overflow worker is canceled and rescheduled on a
different CPU in the same domain. But if the timer is already about to
expire this essentially doubles the interval which might result in a non
detected overflow.

Cancel the overflow worker and reschedule it immediately on a different CPU
in the same domain. The work could be flushed as well, but that would
reschedule it on the same CPU.
```

Example 2:

```
time: POSIX CPU timers: Ensure that variable is initialized

If cpu_timer_sample_group returns -EINVAL, it will not have written into
*sample. Checking for cpu_timer_sample_group's return value precludes the
potential use of an uninitialized value of now in the following block.
Given an invalid clock_idx, the previous code could otherwise overwrite
*oldval in an undefined manner. This is now prevented. We also exploit
short-circuiting of && to sample the timer only if the result will
actually be used to update *oldval.
```

Improved version:

```
posix-cpu-timers: Make set_process_cpu_timer() more robust

Because the return value of cpu_timer_sample_group() is not checked,
compilers and static checkers can legitimately warn about a potential use
of the uninitialized variable 'now'. This is not a runtime issue as all
```

```
    call sites hand in valid clock ids.

    Also cpu_timer_sample_group() is invoked unconditionally even when the
    result is not used because *oldval is NULL.

    Make the invocation conditional and check the return value.
```

Example 3:

```
    The entity can also be used for other purposes.

    Let's rename it to be more generic.
```

Improved version:

```
    The entity can also be used for other purposes.

    Rename it to be more generic.
```

For complex scenarios, especially race conditions and memory ordering issues, it is valuable to depict the scenario with a table which shows the parallelism and the temporal order of events. Here is an example:

```
CPU0                            CPU1
free_irq(X)                     interrupt X
                                spin_lock(desc->lock)
                                wake_irq_thread()
                                spin_unlock(desc->lock)
spin_lock(desc->lock)
remove_action()
shutdown_irq()
release_resources()             thread_handler()
spin_unlock(desc->lock)           access released resources.
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^
synchronize_irq()
```

Lockdep provides similar useful output to depict a possible deadlock scenario:

```
CPU0                                    CPU1
rtmutex_lock(&rcu->rt_mutex)
  spin_lock(&rcu->rt_mutex.wait_lock)
                                        local_irq_disable()
                                        spin_lock(&timer->it_lock)
                                        spin_lock(&rcu->mutex.wait_lock)
--> Interrupt
    spin_lock(&timer->it_lock)
```

Function references in changelogs

When a function is mentioned in the changelog, either the text body or the subject line, please use the format 'function_name()'. Omitting the brackets after the function name can be ambiguous:

```
Subject: subsys/component: Make reservation_count static

reservation_count is only used in reservation_stats. Make it static.
```

The variant with brackets is more precise:

```
Subject: subsys/component: Make reservation_count() static

reservation_count() is only called from reservation_stats(). Make it
static.
```

Backtraces in changelogs

See :ref:`backtraces`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master)(Documentation)(process)maintainer-tip.rst, line 265); *backlink*
>
> Unknown interpreted text role "ref".

Ordering of commit tags

To have a uniform view of the commit tags, the tip maintainers use the following tag ordering scheme:

- Fixes: 12char-SHA1 ("sub/sys: Original subject line")

  A Fixes tag should be added even for changes which do not need to be backported to stable kernels, i.e. when

addressing a recently introduced issue which only affects tip or the current head of mainline. These tags are helpful to identify the original commit and are much more valuable than prominently mentioning the commit which introduced a problem in the text of the changelog itself because they can be automatically extracted.

The following example illustrates the difference:

```
Commit

  abcdef012345678 ("x86/xxx: Replace foo with bar")

left an unused instance of variable foo around. Remove it.

Signed-off-by: J.Dev <j.dev@mail>
```

Please say instead:

```
The recent replacement of foo with bar left an unused instance of
variable foo around. Remove it.

Fixes: abcdef012345678 ("x86/xxx: Replace foo with bar")
Signed-off-by: J.Dev <j.dev@mail>
```

The latter puts the information about the patch into the focus and amends it with the reference to the commit which introduced the issue rather than putting the focus on the original commit in the first place.

- Reported-by: `Reporter <reporter@mail>`
- Originally-by: `Original author <original-author@mail>`
- Suggested-by: `Suggester <suggester@mail>`
- Co-developed-by: `Co-author <co-author@mail>`

  Signed-off: `Co-author <co-author@mail>`

  Note, that Co-developed-by and Signed-off-by of the co-author(s) must come in pairs.
- Signed-off-by: `Author <author@mail>`

  The first Signed-off-by (SOB) after the last Co-developed-by/SOB pair is the author SOB, i.e. the person flagged as author by git.
- Signed-off-by: `Patch handler <handler@mail>`

  SOBs after the author SOB are from people handling and transporting the patch, but were not involved in development. SOB chains should reflect the **real** route a patch took as it was propagated to us, with the first SOB entry signalling primary authorship of a single author. Acks should be given as Acked-by lines and review approvals as Reviewed-by lines.

  If the handler made modifications to the patch or the changelog, then this should be mentioned **after** the changelog text and **above** all commit tags in the following format:

  ```
  ... changelog text ends.

  [ handler: Replaced foo by bar and updated changelog ]

  First-tag: .....
  ```

  Note the two empty new lines which separate the changelog text and the commit tags from that notice.

  If a patch is sent to the mailing list by a handler then the author has to be noted in the first line of the changelog with:

  ```
  From: Author <author@mail>

  Changelog text starts here....
  ```

  so the authorship is preserved. The 'From:' line has to be followed by a empty newline. If that 'From:' line is missing, then the patch would be attributed to the person who sent (transported, handled) it. The 'From:' line is automatically removed when the patch is applied and does not show up in the final git changelog. It merely affects the authorship information of the resulting Git commit.
- Tested-by: `Tester <tester@mail>`
- Reviewed-by: `Reviewer <reviewer@mail>`
- Acked-by: `Acker <acker@mail>`
- Cc: `cc-ed-person <person@mail>`

  If the patch should be backported to stable, then please add a '`Cc: stable@vger.kernel.org`' tag, but do not Cc stable when sending your mail.
- Link: `https://link/to/information`

For referring to an email on LKML or other kernel mailing lists, please use the lore.kernel.org redirector URL:

```
https://lore.kernel.org/r/email-message@id
```

The kernel.org redirector is considered a stable URL, unlike other email archives.

Maintainers will add a Link tag referencing the email of the patch submission when they apply a patch to the tip tree. This tag is useful for later reference and is also used for commit notifications.

Please do not use combined tags, e.g. `Reported-and-tested-by`, as they just complicate automated extraction of tags.

Links to documentation

Providing links to documentation in the changelog is a great help to later debugging and analysis. Unfortunately, URLs often break very quickly because companies restructure their websites frequently. Non-'volatile' exceptions include the Intel SDM and the AMD APM.

Therefore, for 'volatile' documents, please create an entry in the kernel bugzilla https://bugzilla.kernel.org and attach a copy of these documents to the bugzilla entry. Finally, provide the URL of the bugzilla entry in the changelog.

Patch resend or reminders

See :ref:`resend_reminders`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master)(Documentation)(process)maintainer-tip.rst`, line 405); *backlink*

Unknown interpreted text role "ref".

Merge window

Please do not expect large patch series to be handled during the merge window or even during the week before. Such patches should be submitted in mergeable state *at least* a week before the merge window opens. Exceptions are made for bug fixes and *sometimes* for small standalone drivers for new hardware or minimally invasive patches for hardware enablement.

During the merge window, the maintainers instead focus on following the upstream changes, fixing merge window fallout, collecting bug fixes, and allowing themselves a breath. Please respect that.

The release candidate -rc1 is the starting point for new patches to be applied which are targeted for the next merge window.

Git

The tip maintainers accept git pull requests from maintainers who provide subsystem changes for aggregation in the tip tree.

Pull requests for new patch submissions are usually not accepted and do not replace proper patch submission to the mailing list. The main reason for this is that the review workflow is email based.

If you submit a larger patch series it is helpful to provide a git branch in a private repository which allows interested people to easily pull the series for testing. The usual way to offer this is a git URL in the cover letter of the patch series.

Coding style notes

Comment style

Sentences in comments start with an uppercase letter.

Single line comments:

```
/* This is a single line comment */
```

Multi-line comments:

```
/*
 * This is a properly formatted
 * multi-line comment.
 *
 * Larger multi-line comments should be split into paragraphs.
 */
```

No tail comments:

Please refrain from using tail comments. Tail comments disturb the reading flow in almost all contexts, but especially in code:

```
if (somecondition_is_true) /* Don't put a comment here */
        dostuff(); /* Neither here */

seed = MAGIC_CONSTANT; /* Nor here */
```

Use freestanding comments instead:

```
/* This condition is not obvious without a comment */
if (somecondition_is_true) {
        /* This really needs to be documented */
        dostuff();
}

/* This magic initialization needs a comment. Maybe not? */
seed = MAGIC_CONSTANT;
```

Comment the important things:

Comments should be added where the operation is not obvious. Documenting the obvious is just a distraction:

```
/* Decrement refcount and check for zero */
if (refcount_dec_and_test(&p->refcnt)) {
        do;
        lots;
        of;
        magic;
        things;
}
```

Instead, comments should explain the non-obvious details and document constraints:

```
if (refcount_dec_and_test(&p->refcnt)) {
        /*
         * Really good explanation why the magic things below
         * need to be done, ordering and locking constraints,
         * etc..
         */
        do;
        lots;
        of;
        magic;
        /* Needs to be the last operation because ... */
        things;
}
```

Function documentation comments:

To document functions and their arguments please use kernel-doc format and not free form comments:

```
/**
 * magic_function - Do lots of magic stuff
 * @magic:      Pointer to the magic data to operate on
 * @offset:     Offset in the data array of @magic
 *
 * Deep explanation of mysterious things done with @magic along
 * with documentation of the return values.
 *
 * Note, that the argument descriptors above are arranged
 * in a tabular fashion.
 */
```

This applies especially to globally visible functions and inline functions in public header files. It might be overkill to use kernel-doc format for every (static) function which needs a tiny explanation. The usage of descriptive function names often replaces these tiny comments. Apply common sense as always.

Documenting locking requirements

Documenting locking requirements is a good thing, but comments are not necessarily the best choice. Instead of writing:

```
/* Caller must hold foo->lock */
void func(struct foo *foo)
{
        ...
}
```

Please use:

```
void func(struct foo *foo)
{
        lockdep_assert_held(&foo->lock);
```

```
        ...
}
```

In PROVE_LOCKING kernels, lockdep_assert_held() emits a warning if the caller doesn't hold the lock. Comments can't do that.

Bracket rules

Brackets should be omitted only if the statement which follows 'if', 'for', 'while' etc. is truly a single line:

```
if (foo)
        do_something();
```

The following is not considered to be a single line statement even though C does not require brackets:

```
for (i = 0; i < end; i++)
        if (foo[i])
                do_something(foo[i]);
```

Adding brackets around the outer loop enhances the reading flow:

```
for (i = 0; i < end; i++) {
        if (foo[i])
                do_something(foo[i]);
}
```

Variable declarations

The preferred ordering of variable declarations at the beginning of a function is reverse fir tree order:

```
struct long_struct_name *descriptive_name;
unsigned long foo, bar;
unsigned int tmp;
int ret;
```

The above is faster to parse than the reverse ordering:

```
int ret;
unsigned int tmp;
unsigned long foo, bar;
struct long_struct_name *descriptive_name;
```

And even more so than random ordering:

```
unsigned long foo, bar;
int ret;
struct long_struct_name *descriptive_name;
unsigned int tmp;
```

Also please try to aggregate variables of the same type into a single line. There is no point in wasting screen space:

```
unsigned long a;
unsigned long b;
unsigned long c;
unsigned long d;
```

It's really sufficient to do:

```
unsigned long a, b, c, d;
```

Please also refrain from introducing line splits in variable declarations:

```
struct long_struct_name *descriptive_name = container_of(bar,
                                                         struct long_struct_name,
                                                         member);
struct foobar foo;
```

It's way better to move the initialization to a separate line after the declarations:

```
struct long_struct_name *descriptive_name;
struct foobar foo;

descriptive_name = container_of(bar, struct long_struct_name, member);
```

Variable types

Please use the proper u8, u16, u32, u64 types for variables which are meant to describe hardware or are used as arguments for functions which access hardware. These types are clearly defining the bit width and avoid truncation, expansion and 32/64-bit confusion.

u64 is also recommended in code which would become ambiguous for 32-bit kernels when 'unsigned long' would be used instead.

While in such situations 'unsigned long long' could be used as well, u64 is shorter and also clearly shows that the operation is required to be 64 bits wide independent of the target CPU.

Please use 'unsigned int' instead of 'unsigned'.

Constants

Please do not use literal (hexa)decimal numbers in code or initializers. Either use proper defines which have descriptive names or consider using an enum.

Struct declarations and initializers

Struct declarations should align the struct member names in a tabular fashion:

```
struct bar_order {
        unsigned int    guest_id;
        int             ordered_item;
        struct menu     *menu;
};
```

Please avoid documenting struct members within the declaration, because this often results in strangely formatted comments and the struct members become obfuscated:

```
struct bar_order {
        unsigned int    guest_id; /* Unique guest id */
        int             ordered_item;
        /* Pointer to a menu instance which contains all the drinks */
        struct menu     *menu;
};
```

Instead, please consider using the kernel-doc format in a comment preceding the struct declaration, which is easier to read and has the added advantage of including the information in the kernel documentation, for example, as follows:

```
/**
 * struct bar_order - Description of a bar order
 * @guest_id:           Unique guest id
 * @ordered_item:       The item number from the menu
 * @menu:               Pointer to the menu from which the item
 *                      was ordered
 *
 * Supplementary information for using the struct.
 *
 * Note, that the struct member descriptors above are arranged
 * in a tabular fashion.
 */
struct bar_order {
        unsigned int    guest_id;
        int             ordered_item;
        struct menu     *menu;
};
```

Static struct initializers must use C99 initializers and should also be aligned in a tabular fashion:

```
static struct foo statfoo = {
        .a              = 0,
        .plain_integer  = CONSTANT_DEFINE_OR_ENUM,
        .bar            = &statbar,
};
```

Note that while C99 syntax allows the omission of the final comma, we recommend the use of a comma on the last line because it makes reordering and addition of new lines easier, and makes such future patches slightly easier to read as well.

Line breaks

Restricting line length to 80 characters makes deeply indented code hard to read. Consider breaking out code into helper functions to avoid excessive line breaking.

The 80 character rule is not a strict rule, so please use common sense when breaking lines. Especially format strings should never be broken up.

When splitting function declarations or function calls, then please align the first argument in the second line with the first argument in the first line:

```
static int long_function_name(struct foobar *barfoo, unsigned int id,
                              unsigned int offset)
{

      if (!id) {
              ret = longer_function_name(barfoo, DEFAULT_BARFOO_ID,
```

```
                                        offset);
        ...
```

Namespaces

Function/variable namespaces improve readability and allow easy grepping. These namespaces are string prefixes for globally visible function and variable names, including inlines. These prefixes should combine the subsystem and the component name such as 'x86_comp_', 'sched_', 'irq_', and 'mutex_'.

This also includes static file scope functions that are immediately put into globally visible driver templates - it's useful for those symbols to carry a good prefix as well, for backtrace readability.

Namespace prefixes may be omitted for local static functions and variables. Truly local functions, only called by other local functions, can have shorter descriptive names - our primary concern is greppability and backtrace readability.

Please note that 'xxx_vendor_' and 'vendor_xxx_` prefixes are not helpful for static functions in vendor-specific files. After all, it is already clear that the code is vendor-specific. In addition, vendor names should only be for truly vendor-specific functionality.

As always apply common sense and aim for consistency and readability.

Commit notifications

The tip tree is monitored by a bot for new commits. The bot sends an email for each new commit to a dedicated mailing list (`linux-tip-commits@vger.kernel.org`) and Cc's all people who are mentioned in one of the commit tags. It uses the email message ID from the Link tag at the end of the tag list to set the In-Reply-To email header so the message is properly threaded with the patch submission email.

The tip maintainers and submaintainers try to reply to the submitter when merging a patch, but they sometimes forget or it does not fit the workflow of the moment. While the bot message is purely mechanical, it also implies a 'Thank you! Applied.'.