Linux Core-api Documentation

The kernel development community

Jun 10, 2024

CONTENTS

This is the beginning of a manual for core kernel APIs. The conversion (and writing!) of documents for this manual is much appreciated!

CHAPTER

ONE

CORE UTILITIES

This section has general and "core core" documentation. The first is a massive grab-bag of kerneldoc info left over from the docbook days; it should really be broken up someday when somebody finds the energy to do it.

1.1 The Linux Kernel API

1.1.1 List Management Functions

void **INIT_LIST_HEAD**(struct list_head *list)

Initialize a list_head structure

Parameters

struct list_head *list

list_head structure to be initialized.

Description

Initializes the list_head to point to itself. If it is a list header, the result is an empty list.

void **list_add**(struct list_head *new, struct list_head *head)

add a new entry

Parameters

struct list_head *new

new entry to be added

struct list_head *head

list head to add it after

Description

Insert a new entry after the specified head. This is good for implementing stacks.

void **list_add_tail**(struct list_head *new, struct list_head *head)

add a new entry

Parameters

struct list_head *new

new entry to be added

struct list_head *head
list head to add it before

Description

Insert a new entry before the specified head. This is useful for implementing queues.

void **list_del**(struct list_head *entry)
deletes entry from list.

Parameters

struct list_head *entry
the element to delete from the list.

Note

list_empty() on entry does not return true after this, the entry is in an undefined state.

void **list_replace**(struct list_head *old, struct list_head *new)
replace old entry by new one

Parameters

struct list_head *old
the element to be replaced

struct list_head *new
the new element to insert

Description

If **old** was empty, it will be overwritten.

void **list_replace_init**(struct list_head *old, struct list_head *new)
replace old entry by new one and initialize the old one

Parameters

struct list_head *old
the element to be replaced

struct list_head *new
the new element to insert

Description

If **old** was empty, it will be overwritten.

void **list_swap**(struct list_head *entry1, struct list_head *entry2)
replace entry1 with entry2 and re-add entry1 at entry2's position

Parameters

struct list_head *entry1
the location to place entry2

struct list_head *entry2
the location to place entry1

void **list_del_init**(struct list_head *entry)
deletes entry from list and reinitialize it.

Parameters

struct list_head *entry
the element to delete from the list.

void **list_move**(struct list_head *list, struct list_head *head)
delete from one list and add as another's head

Parameters

struct list_head *list
the entry to move

struct list_head *head
the head that will precede our entry

void **list_move_tail**(struct list_head *list, struct list_head *head)
delete from one list and add as another's tail

Parameters

struct list_head *list
the entry to move

struct list_head *head
the head that will follow our entry

void **list_bulk_move_tail**(struct list_head *head, struct list_head *first, struct list_head *last)
move a subsection of a list to its tail

Parameters

struct list_head *head
the head that will follow our entry

struct list_head *first
first entry to move

struct list_head *last
last entry to move, can be the same as first

Description

Move all entries between **first** and including **last** before **head**. All three entries must belong to the same linked list.

int **list_is_first**(const struct list_head *list, const struct list_head *head)

- tests whether **list** is the first entry in list **head**

Parameters

const struct list_head *list
the entry to test

const struct list_head *head
the head of the list

int **list_is_last**(const struct list_head *list, const struct list_head *head)
tests whether **list** is the last entry in list **head**

Parameters

const struct list_head *list
the entry to test

const struct list_head *head
the head of the list

int **list_is_head**(const struct list_head *list, const struct list_head *head)
tests whether **list** is the list **head**

Parameters

const struct list_head *list
the entry to test

const struct list_head *head
the head of the list

int **list_empty**(const struct list_head *head)
tests whether a list is empty

Parameters

const struct list_head *head
the list to test.

void **list_del_init_careful**(struct list_head *entry)
deletes entry from list and reinitialize it.

Parameters

struct list_head *entry
the element to delete from the list.

Description

This is the same as *list_del_init()*, except designed to be used together with *list_empty_careful()* in a way to guarantee ordering of other memory operations.

Any memory operations done before a *list_del_init_careful()* are guaranteed to be visible after a *list_empty_careful()* test.

int **list_empty_careful**(const struct list_head *head)
tests whether a list is empty and not being modified

Parameters

const struct list_head *head
the list to test

Description

tests whether a list is empty _and_ checks that no other CPU might be in the process of modifying either member (next or prev)

NOTE

using *list_empty_careful()* without synchronization can only be safe if the only activity that can happen to the list entry is *list_del_init()*. Eg. it cannot be used if another CPU could re-list_add() it.

void **list_rotate_left**(struct list_head *head)

rotate the list to the left

Parameters

struct list_head *head
the head of the list

void **list_rotate_to_front**(struct list_head *list, struct list_head *head)

Rotate list to specific item.

Parameters

struct list_head *list
The desired new front of the list.

struct list_head *head
The head of the list.

Description

Rotates list so that **list** becomes the new front of the list.

int **list_is_singular**(const struct list_head *head)

tests whether a list has just one entry.

Parameters

const struct list_head *head
the list to test.

void **list_cut_position**(struct list_head *list, struct list_head *head, struct list_head *entry)

cut a list into two

Parameters

struct list_head *list
a new list to add all removed entries

struct list_head *head
a list with entries

struct list_head *entry
an entry within head, could be the head itself and if so we won't cut the list

Description

This helper moves the initial part of **head**, up to and including **entry**, from **head** to **list**. You should pass on **entry** an element you know is on **head**. **list** should be an empty list or a list you do not care about losing its data.

void **list_cut_before**(struct list_head *list, struct list_head *head, struct list_head *entry)

cut a list into two, before given entry

Parameters

struct list_head *list
  a new list to add all removed entries

struct list_head *head
  a list with entries

struct list_head *entry
  an entry within head, could be the head itself

Description

This helper moves the initial part of **head**, up to but excluding **entry**, from **head** to **list**. You should pass in **entry** an element you know is on **head**. **list** should be an empty list or a list you do not care about losing its data. If **entry** == **head**, all entries on **head** are moved to **list**.

void **list_splice**(const struct list_head *list, struct list_head *head)
  join two lists, this is designed for stacks

Parameters

const struct list_head *list
  the new list to add.

struct list_head *head
  the place to add it in the first list.

void **list_splice_tail**(struct list_head *list, struct list_head *head)
  join two lists, each list being a queue

Parameters

struct list_head *list
  the new list to add.

struct list_head *head
  the place to add it in the first list.

void **list_splice_init**(struct list_head *list, struct list_head *head)
  join two lists and reinitialise the emptied list.

Parameters

struct list_head *list
  the new list to add.

struct list_head *head
  the place to add it in the first list.

Description

The list at **list** is reinitialised

void **list_splice_tail_init**(struct list_head *list, struct list_head *head)
  join two lists and reinitialise the emptied list

Parameters

struct list_head *list
  the new list to add.

struct list_head *head
  the place to add it in the first list.

Description

Each of the lists is a queue. The list at **list** is reinitialised

list_entry

`list_entry (ptr, type, member)`

get the struct for this entry

Parameters

ptr
the `struct list_head` pointer.

type
the type of the struct this is embedded in.

member
the name of the list_head within the struct.

list_first_entry

`list_first_entry (ptr, type, member)`

get the first element from a list

Parameters

ptr
the list head to take the element from.

type
the type of the struct this is embedded in.

member
the name of the list_head within the struct.

Description

Note, that list is expected to be not empty.

list_last_entry

`list_last_entry (ptr, type, member)`

get the last element from a list

Parameters

ptr
the list head to take the element from.

type
the type of the struct this is embedded in.

member
the name of the list_head within the struct.

Description

Note, that list is expected to be not empty.

list_first_entry_or_null

`list_first_entry_or_null (ptr, type, member)`

get the first element from a list

Parameters

ptr
the list head to take the element from.

type
the type of the struct this is embedded in.

member
the name of the list_head within the struct.

Description

Note that if the list is empty, it returns NULL.

list_next_entry

`list_next_entry (pos, member)`

get the next element in list

Parameters

pos
the type * to cursor

member
the name of the list_head within the struct.

list_next_entry_circular

`list_next_entry_circular (pos, head, member)`

get the next element in list

Parameters

pos
the type * to cursor.

head
the list head to take the element from.

member
the name of the list_head within the struct.

Description

Wraparound if pos is the last element (return the first element). Note, that list is expected to be not empty.

list_prev_entry

`list_prev_entry (pos, member)`

get the prev element in list

Parameters

pos
the type * to cursor

member
the name of the list_head within the struct.

list_prev_entry_circular

`list_prev_entry_circular (pos, head, member)`

get the prev element in list

Parameters

pos
the type * to cursor.

head
the list head to take the element from.

member
the name of the list_head within the struct.

Description

Wraparound if pos is the first element (return the last element). Note, that list is expected to be not empty.

list_for_each

`list_for_each (pos, head)`

iterate over a list

Parameters

pos
the `struct list_head` to use as a loop cursor.

head
the head for your list.

list_for_each_rcu

`list_for_each_rcu (pos, head)`

Iterate over a list in an RCU-safe fashion

Parameters

pos
the `struct list_head` to use as a loop cursor.

head
the head for your list.

list_for_each_continue

`list_for_each_continue (pos, head)`

continue iteration over a list

Parameters

pos
: the `struct list_head` to use as a loop cursor.

head
: the head for your list.

Description

Continue to iterate over a list, continuing after the current position.

list_for_each_prev

`list_for_each_prev (pos, head)`

iterate over a list backwards

Parameters

pos
: the `struct list_head` to use as a loop cursor.

head
: the head for your list.

list_for_each_safe

`list_for_each_safe (pos, n, head)`

iterate over a list safe against removal of list entry

Parameters

pos
: the `struct list_head` to use as a loop cursor.

n
: another `struct list_head` to use as temporary storage

head
: the head for your list.

list_for_each_prev_safe

`list_for_each_prev_safe (pos, n, head)`

iterate over a list backwards safe against removal of list entry

Parameters

pos
: the `struct list_head` to use as a loop cursor.

n
: another `struct list_head` to use as temporary storage

head
: the head for your list.

size_t **list_count_nodes**(struct list_head *head)

count nodes in the list

Parameters

struct list_head *head
the head for your list.

list_entry_is_head

`list_entry_is_head (pos, head, member)`

test if the entry points to the head of the list

Parameters

pos
the type * to cursor

head
the head for your list.

member
the name of the list_head within the struct.

list_for_each_entry

`list_for_each_entry (pos, head, member)`

iterate over list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

list_for_each_entry_reverse

`list_for_each_entry_reverse (pos, head, member)`

iterate backwards over list of given type.

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

list_prepare_entry

`list_prepare_entry (pos, head, member)`

prepare a pos entry for use in *list_for_each_entry_continue()*

Parameters

pos
the type * to use as a start point

head
the head of the list

member
the name of the list_head within the struct.

Description

Prepares a pos entry for use as a start point in *list_for_each_entry_continue()*.

list_for_each_entry_continue

`list_for_each_entry_continue (pos, head, member)`

continue iteration over list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

Description

Continue to iterate over list of given type, continuing after the current position.

list_for_each_entry_continue_reverse

`list_for_each_entry_continue_reverse (pos, head, member)`

iterate backwards from the given point

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

Description

Start to iterate over list of given type backwards, continuing after the current position.

list_for_each_entry_from

`list_for_each_entry_from (pos, head, member)`

iterate over list of given type from the current point

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

Description

Iterate over list of given type, continuing from current position.

list_for_each_entry_from_reverse

`list_for_each_entry_from_reverse (pos, head, member)`

iterate backwards over list of given type from the current point

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

Description

Iterate backwards over list of given type, continuing from current position.

list_for_each_entry_safe

`list_for_each_entry_safe (pos, n, head, member)`

iterate over list of given type safe against removal of list entry

Parameters

pos
the type * to use as a loop cursor.

n
another type * to use as temporary storage

head
the head for your list.

member
the name of the list_head within the struct.

list_for_each_entry_safe_continue

`list_for_each_entry_safe_continue (pos, n, head, member)`

continue list iteration safe against removal

Parameters

pos
the type * to use as a loop cursor.

n
another type * to use as temporary storage

head
the head for your list.

member
the name of the list_head within the struct.

Description

Iterate over list of given type, continuing after current point, safe against removal of list entry.

list_for_each_entry_safe_from

`list_for_each_entry_safe_from (pos, n, head, member)`

iterate over list from current point safe against removal

Parameters

pos
the type * to use as a loop cursor.

n
another type * to use as temporary storage

head
the head for your list.

member
the name of the list_head within the struct.

Description

Iterate over list of given type from current point, safe against removal of list entry.

list_for_each_entry_safe_reverse

`list_for_each_entry_safe_reverse (pos, n, head, member)`

iterate backwards over list safe against removal

Parameters

pos
the type * to use as a loop cursor.

n
another type * to use as temporary storage

head
the head for your list.

member
the name of the list_head within the struct.

Description

Iterate backwards over list of given type, safe against removal of list entry.

list_safe_reset_next

`list_safe_reset_next (pos, n, member)`

reset a stale list_for_each_entry_safe loop

Parameters

pos
the loop cursor used in the list_for_each_entry_safe loop

n
temporary storage used in list_for_each_entry_safe

member
the name of the list_head within the struct.

Description

list_safe_reset_next is not safe to use in general if the list may be modified concurrently (eg. the lock is dropped in the loop body). An exception to this is if the cursor element (pos) is pinned in the list, and list_safe_reset_next is called after re-taking the lock and before completing the current iteration of the loop body.

int **hlist_unhashed**(const struct hlist_node *h)

Has node been removed from list and reinitialized?

Parameters

const struct hlist_node *h
Node to be checked

Description

Not that not all removal functions will leave a node in unhashed state. For example, *hlist_nulls_del_init_rcu()* does leave the node in unhashed state, but hlist_nulls_del() does not.

int **hlist_unhashed_lockless**(const struct hlist_node *h)

Version of hlist_unhashed for lockless use

Parameters

const struct hlist_node *h
Node to be checked

Description

This variant of *hlist_unhashed()* must be used in lockless contexts to avoid potential load-tearing. The READ_ONCE() is paired with the various WRITE_ONCE() in hlist helpers that are defined below.

int **hlist_empty**(const struct hlist_head *h)

Is the specified hlist_head structure an empty hlist?

Parameters

const struct hlist_head *h
Structure to check.

void **hlist_del**(struct hlist_node *n)
Delete the specified hlist_node from its list

Parameters

struct hlist_node *n
Node to delete.

Description

Note that this function leaves the node in hashed state. Use *hlist_del_init()* or similar instead to unhash **n**.

void **hlist_del_init**(struct hlist_node *n)
Delete the specified hlist_node from its list and initialize

Parameters

struct hlist_node *n
Node to delete.

Description

Note that this function leaves the node in unhashed state.

void **hlist_add_head**(struct hlist_node *n, struct hlist_head *h)
add a new entry at the beginning of the hlist

Parameters

struct hlist_node *n
new entry to be added

struct hlist_head *h
hlist head to add it after

Description

Insert a new entry after the specified head. This is good for implementing stacks.

void **hlist_add_before**(struct hlist_node *n, struct hlist_node *next)
add a new entry before the one specified

Parameters

struct hlist_node *n
new entry to be added

struct hlist_node *next
hlist node to add it before, which must be non-NULL

void **hlist_add_behind**(struct hlist_node *n, struct hlist_node *prev)
add a new entry after the one specified

Parameters

struct hlist_node *n
new entry to be added

struct hlist_node *prev
hlist node to add it after, which must be non-NULL

void **hlist_add_fake**(struct hlist_node *n)

create a fake hlist consisting of a single headless node

Parameters

struct hlist_node *n

Node to make a fake list out of

Description

This makes **n** appear to be its own predecessor on a headless hlist. The point of this is to allow things like *hlist_del()* to work correctly in cases where there is no list.

bool **hlist_fake**(struct hlist_node *h)

Is this node a fake hlist?

Parameters

struct hlist_node *h

Node to check for being a self-referential fake hlist.

bool **hlist_is_singular_node**(struct hlist_node *n, struct hlist_head *h)

is node the only element of the specified hlist?

Parameters

struct hlist_node *n

Node to check for singularity.

struct hlist_head *h

Header for potentially singular list.

Description

Check whether the node is the only node of the head without accessing head, thus avoiding unnecessary cache misses.

void **hlist_move_list**(struct hlist_head *old, struct hlist_head *new)

Move an hlist

Parameters

struct hlist_head *old

hlist_head for old list.

struct hlist_head *new

hlist_head for new list.

Description

Move a list from one list head to another. Fixup the pprev reference of the first entry if it exists.

hlist_for_each_entry

```
hlist_for_each_entry (pos, head, member)
```

iterate over list of given type

Parameters

pos

the type * to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_node within the struct.

hlist_for_each_entry_continue

`hlist_for_each_entry_continue (pos, member)`

iterate over a hlist continuing after current point

Parameters

pos
the type * to use as a loop cursor.

member
the name of the hlist_node within the struct.

hlist_for_each_entry_from

`hlist_for_each_entry_from (pos, member)`

iterate over a hlist continuing from current point

Parameters

pos
the type * to use as a loop cursor.

member
the name of the hlist_node within the struct.

hlist_for_each_entry_safe

`hlist_for_each_entry_safe (pos, n, head, member)`

iterate over list of given type safe against removal of list entry

Parameters

pos
the type * to use as a loop cursor.

n
a `struct hlist_node` to use as temporary storage

head
the head for your list.

member
the name of the hlist_node within the struct.

1.1.2 Basic C Library Functions

When writing drivers, you cannot in general use routines which are from the C Library. Some of the functions have been found generally useful and they are listed below. The behaviour of these functions may vary slightly from those defined by ANSI, and these deviations are noted in the text.

String Conversions

unsigned long long **simple_strtoull**(const char *cp, char **endp, unsigned int base)
convert a string to an unsigned long long

Parameters

const char *cp
The start of the string

char **endp
A pointer to the end of the parsed string will be placed here

unsigned int base
The number base to use

Description

This function has caveats. Please use kstrtoull instead.

unsigned long **simple_strtoul**(const char *cp, char **endp, unsigned int base)
convert a string to an unsigned long

Parameters

const char *cp
The start of the string

char **endp
A pointer to the end of the parsed string will be placed here

unsigned int base
The number base to use

Description

This function has caveats. Please use kstrtoul instead.

long **simple_strtol**(const char *cp, char **endp, unsigned int base)
convert a string to a signed long

Parameters

const char *cp
The start of the string

char **endp
A pointer to the end of the parsed string will be placed here

unsigned int base
The number base to use

Description

This function has caveats. Please use kstrtol instead.

long long **simple_strtoll**(const char *cp, char **endp, unsigned int base)
convert a string to a signed long long

Parameters

const char *cp
The start of the string

char **endp
A pointer to the end of the parsed string will be placed here

unsigned int base
The number base to use

Description

This function has caveats. Please use kstrtoll instead.

int **vsnprintf**(char *buf, size_t size, const char *fmt, va_list args)
Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

size_t size
The size of the buffer, including the trailing null space

const char *fmt
The format string to use

va_list args
Arguments for the format string

Description

This function generally follows C99 vsnprintf, but has some extensions and a few limitations:

- ``n`` is unsupported
- ``p``* is handled by pointer()

See pointer() or *How to get printk format specifiers right* for more extensive description.

Please update the documentation in both places when making changes

The return value is the number of characters which would be generated for the given input, excluding the trailing '0', as per ISO C99. If you want to have the exact number of characters written into **buf** as return value (not including the trailing '0'), use *vscnprintf()*. If the return is greater than or equal to **size**, the resulting string is truncated.

If you're not already dealing with a va_list consider using *snprintf()*.

int **vscnprintf**(char *buf, size_t size, const char *fmt, va_list args)
Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

size_t size
The size of the buffer, including the trailing null space

const char *fmt
The format string to use

va_list args
Arguments for the format string

Description

The return value is the number of characters which have been written into the **buf** not including the trailing '0'. If **size** is == 0 the function returns 0.

If you're not already dealing with a va_list consider using *scnprintf()*.

See the vsnprintf() documentation for format string extensions over C99.

int **snprintf**(char *buf, size_t size, const char *fmt, ...)
Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

size_t size
The size of the buffer, including the trailing null space

const char *fmt
The format string to use

...
Arguments for the format string

Description

The return value is the number of characters which would be generated for the given input, excluding the trailing null, as per ISO C99. If the return is greater than or equal to **size**, the resulting string is truncated.

See the vsnprintf() documentation for format string extensions over C99.

int **scnprintf**(char *buf, size_t size, const char *fmt, ...)
Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

size_t size
The size of the buffer, including the trailing null space

const char *fmt
The format string to use

...
Arguments for the format string

Description

The return value is the number of characters written into **buf** not including the trailing '0'. If **size** is == 0 the function returns 0.

int **vsprintf**(char *buf, const char *fmt, va_list args)

Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

const char *fmt
The format string to use

va_list args
Arguments for the format string

Description

The function returns the number of characters written into **buf**. Use vsnprintf() or *vscnprintf()* in order to avoid buffer overflows.

If you're not already dealing with a va_list consider using *sprintf()*.

See the vsnprintf() documentation for format string extensions over C99.

int **sprintf**(char *buf, const char *fmt, ...)

Format a string and place it in a buffer

Parameters

char *buf
The buffer to place the result into

const char *fmt
The format string to use

...
Arguments for the format string

Description

The function returns the number of characters written into **buf**. Use *snprintf()* or *scnprintf()* in order to avoid buffer overflows.

See the vsnprintf() documentation for format string extensions over C99.

int **vbin_printf**(u32 *bin_buf, size_t size, const char *fmt, va_list args)

Parse a format string and place args' binary value in a buffer

Parameters

u32 *bin_buf
The buffer to place args' binary value

size_t size
The size of the buffer(by words(32bits), not characters)

const char *fmt
The format string to use

va_list args
Arguments for the format string

Description

The format follows C99 vsnprintf, except n is ignored, and its argument is skipped.

The return value is the number of words(32bits) which would be generated for the given input.

NOTE

If the return value is greater than **size**, the resulting bin_buf is NOT valid for *bstr_printf()*.

int **bstr_printf**(char *buf, size_t size, const char *fmt, const u32 *bin_buf)
Format a string from binary arguments and place it in a buffer

Parameters

char *buf
The buffer to place the result into

size_t size
The size of the buffer, including the trailing null space

const char *fmt
The format string to use

const u32 *bin_buf
Binary arguments for the format string

Description

This function like C99 vsnprintf, but the difference is that vsnprintf gets arguments from stack, and bstr_printf gets arguments from **bin_buf** which is a binary buffer that generated by vbin_printf.

The format follows C99 vsnprintf, but has some extensions:
see vsnprintf comment for details.

The return value is the number of characters which would be generated for the given input, excluding the trailing '0', as per ISO C99. If you want to have the exact number of characters written into **buf** as return value (not including the trailing '0'), use *vscnprintf()*. If the return is greater than or equal to **size**, the resulting string is truncated.

int **bprintf**(u32 *bin_buf, size_t size, const char *fmt, ...)
Parse a format string and place args' binary value in a buffer

Parameters

u32 *bin_buf
The buffer to place args' binary value

size_t size
The size of the buffer(by words(32bits), not characters)

const char *fmt
The format string to use

...
Arguments for the format string

Description

The function returns the number of words(u32) written into **bin_buf**.

int **vsscanf**(const char *buf, const char *fmt, va_list args)

Unformat a buffer into a list of arguments

Parameters

const char *buf
input buffer

const char *fmt
format of buffer

va_list args
arguments

int **sscanf**(const char *buf, const char *fmt, ...)

Unformat a buffer into a list of arguments

Parameters

const char *buf
input buffer

const char *fmt
formatting of buffer

...
resulting arguments

int **kstrtoul**(const char *s, unsigned int base, unsigned long *res)

convert a string to an unsigned long

Parameters

const char *s
The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.

unsigned int base
The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

unsigned long *res
Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtoul(). Return code must be checked.

int **kstrtol**(const char *s, unsigned int base, long *res)

convert a string to a long

Parameters

const char *s
: The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.

unsigned int base
: The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

long *res
: Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtol(). Return code must be checked.

int **kstrtoull**(const char *s, unsigned int base, unsigned long long *res)
: convert a string to an unsigned long long

Parameters

const char *s
: The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.

unsigned int base
: The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

unsigned long long *res
: Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtoull(). Return code must be checked.

int **kstrtoll**(const char *s, unsigned int base, long long *res)
: convert a string to a long long

Parameters

const char *s
: The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.

unsigned int base
: The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

long long *res
Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtoll(). Return code must be checked.

int **kstrtouint**(const char *s, unsigned int base, unsigned int *res)
convert a string to an unsigned int

Parameters

const char *s
The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.

unsigned int base
The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

unsigned int *res
Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtoul(). Return code must be checked.

int **kstrtoint**(const char *s, unsigned int base, int *res)
convert a string to an int

Parameters

const char *s
The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.

unsigned int base
The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

int *res
Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Preferred over simple_strtol(). Return code must be checked.

int **kstrtobool**(const char *s, bool *res)
convert common user inputs into boolean values

Parameters

const char *s
input string

bool *res
result

Description

This routine returns 0 iff the first character is one of 'YyTt1NnFf0', or [oO][NnFf] for "on" and "off". Otherwise it will return -EINVAL. Value pointed to by res is updated upon finding a match.

void **string_get_size**(u64 size, u64 blk_size, const enum string_size_units units, char *buf, int len)

get the size in the specified units

Parameters

u64 size
The size to be converted in blocks

u64 blk_size
Size of the block (use 1 for size in bytes)

const enum string_size_units units
units to use (powers of 1000 or 1024)

char *buf
buffer to format to

int len
length of buffer

Description

This function returns a string formatted to 3 significant figures giving the size in the required units. **buf** should have room for at least 9 bytes and will always be zero terminated.

int **parse_int_array_user**(const char __user *from, size_t count, int **array)

Split string into a sequence of integers

Parameters

const char __user *from
The user space buffer to read from

size_t count
The maximum number of bytes to read

int **array
Returned pointer to sequence of integers

Description

On success **array** is allocated and initialized with a sequence of integers extracted from the **from** plus an additional element that begins the sequence and specifies the integers count.

Caller takes responsibility for freeing **array** when it is no longer needed.

int **string_unescape**(char *src, char *dst, size_t size, unsigned int flags)

unquote characters in the given string

Parameters

char *src
    source buffer (escaped)

char *dst
    destination buffer (unescaped)

size_t size
    size of the destination buffer (0 to unlimit)

unsigned int flags
    combination of the flags.

Description

The function unquotes characters in the given string.

Because the size of the output will be the same as or less than the size of the input, the transformation may be performed in place.

Caller must provide valid source and destination pointers. Be aware that destination buffer will always be NULL-terminated. Source string must be NULL-terminated as well. The supported flags are:

```
UNESCAPE_SPACE:
        '\f' - form feed
        '\n' - new line
        '\r' - carriage return
        '\t' - horizontal tab
        '\v' - vertical tab
UNESCAPE_OCTAL:
        '\NNN' - byte with octal value NNN (1 to 3 digits)
UNESCAPE_HEX:
        '\xHH' - byte with hexadecimal value HH (1 to 2 digits)
UNESCAPE_SPECIAL:
        '\"' - double quote
        '\\' - backslash
        '\a' - alert (BEL)
        '\e' - escape
UNESCAPE_ANY:
        all previous together
```

Return

The amount of the characters processed to the destination buffer excluding trailing '0' is returned.

int **string_escape_mem**(const char *src, size_t isz, char *dst, size_t osz, unsigned int flags, const char *only)

    quote characters in the given memory buffer

Parameters

const char *src
    source buffer (unescaped)

size_t isz
    source buffer size

char *dst
destination buffer (escaped)

size_t osz
destination buffer size

unsigned int flags
combination of the flags

const char *only
NULL-terminated string containing characters used to limit the selected escape class. If characters are included in **only** that would not normally be escaped by the classes selected in **flags**, they will be copied to **dst** unescaped.

Description

The process of escaping byte buffer includes several parts. They are applied in the following sequence.

1. The character is not matched to the one from **only** string and thus must go as-is to the output.
2. The character is matched to the printable and ASCII classes, if asked, and in case of match it passes through to the output.
3. The character is matched to the printable or ASCII class, if asked, and in case of match it passes through to the output.
4. The character is checked if it falls into the class given by **flags**. `ESCAPE_OCTAL` and `ESCAPE_HEX` are going last since they cover any character. Note that they actually can't go together, otherwise `ESCAPE_HEX` will be ignored.

Caller must provide valid source and destination pointers. Be aware that destination buffer will not be NULL-terminated, thus caller have to append it if needs. The supported flags are:

```
%ESCAPE_SPACE: (special white space, not space itself)
        '\f' - form feed
        '\n' - new line
        '\r' - carriage return
        '\t' - horizontal tab
        '\v' - vertical tab
%ESCAPE_SPECIAL:
        '\"' - double quote
        '\\' - backslash
        '\a' - alert (BEL)
        '\e' - escape
%ESCAPE_NULL:
        '\0' - null
%ESCAPE_OCTAL:
        '\NNN' - byte with octal value NNN (3 digits)
%ESCAPE_ANY:
        all previous together
%ESCAPE_NP:
        escape only non-printable characters, checked by isprint()
%ESCAPE_ANY_NP:
        all previous together
```

```
%ESCAPE_HEX:
        '\xHH' - byte with hexadecimal value HH (2 digits)
%ESCAPE_NA:
        escape only non-ascii characters, checked by isascii()
%ESCAPE_NAP:
        escape only non-printable or non-ascii characters
%ESCAPE_APPEND:
        append characters from @only to be escaped by the given classes
```

ESCAPE_APPEND would help to pass additional characters to the escaped, when one of ESCAPE_NP, ESCAPE_NA, or ESCAPE_NAP is provided.

One notable caveat, the ESCAPE_NAP, ESCAPE_NP and ESCAPE_NA have the higher priority than the rest of the flags (ESCAPE_NAP is the highest). It doesn't make much sense to use either of them without ESCAPE_OCTAL or ESCAPE_HEX, because they cover most of the other character classes. ESCAPE_NAP can utilize ESCAPE_SPACE or ESCAPE_SPECIAL in addition to the above.

Return

The total size of the escaped output that would be generated for the given input and flags. To check whether the output was truncated, compare the return value to osz. There is room left in dst for a '0' terminator if and only if ret < osz.

char ****kasprintf_strarray**(gfp_t gfp, const char *prefix, size_t n)

allocate and fill array of sequential strings

Parameters

gfp_t gfp
flags for the slab allocator

const char *prefix
prefix to be used

size_t n
amount of lines to be allocated and filled

Description

Allocates and fills **n** strings using pattern "s-````zu", where prefix is provided by caller. The caller is responsible to free them with *kfree_strarray()* after use.

Returns array of strings or NULL when memory can't be allocated.

void **kfree_strarray**(char **array, size_t n)

free a number of dynamically allocated strings contained in an array and the array itself

Parameters

char **array
Dynamically allocated array of strings to free.

size_t n
Number of strings (starting from the beginning of the array) to free.

Description

Passing a non-NULL **array** and **n** == 0 as well as NULL **array** are valid use-cases. If **array** is NULL, the function does nothing.

ssize_t **strscpy_pad**(char *dest, const char *src, size_t count)
Copy a C-string into a sized buffer

Parameters

char *dest
Where to copy the string to

const char *src
Where to copy the string from

size_t count
Size of destination buffer

Description

Copy the string, or as much of it as fits, into the dest buffer. The behavior is undefined if the string buffers overlap. The destination buffer is always `NUL` terminated, unless it's zero-sized.

If the source string is shorter than the destination buffer, zeros the tail of the destination buffer.

For full explanation of why you may want to consider using the 'strscpy' functions please see the function docstring for strscpy().

Return

- The number of characters copied (not including the trailing `NUL`)
- -E2BIG if count is 0 or **src** was truncated.

char ***skip_spaces**(const char *str)
Removes leading whitespace from **str**.

Parameters

const char *str
The string to be stripped.

Description

Returns a pointer to the first non-whitespace character in **str**.

char ***strim**(char *s)
Removes leading and trailing whitespace from **s**.

Parameters

char *s
The string to be stripped.

Description

Note that the first trailing whitespace is replaced with a `NUL-terminator` in the given string **s**. Returns a pointer to the first non-whitespace character in **s**.

bool **sysfs_streq**(const char *s1, const char *s2)
return true if strings are equal, modulo trailing newline

Parameters

const char *s1
one string

const char *s2
another string

Description

This routine returns true iff two strings are equal, treating both NUL and newline-then-NUL as equivalent string terminations. It's geared for use with sysfs input strings, which generally terminate with newlines but are compared against values without newlines.

int **match_string**(const char *const *array, size_t n, const char *string)
matches given string in an array

Parameters

const char * const *array
array of strings

size_t n
number of strings in the array or -1 for NULL terminated arrays

const char *string
string to match with

Description

This routine will look for a string in an array of strings up to the n-th element in the array or until the first NULL element.

Historically the value of -1 for **n**, was used to search in arrays that are NULL terminated. However, the function does not make a distinction when finishing the search: either **n** elements have been compared OR the first NULL element was found.

Return

index of a **string** in the **array** if matches, or `-EINVAL` otherwise.

int **__sysfs_match_string**(const char *const *array, size_t n, const char *str)
matches given string in an array

Parameters

const char * const *array
array of strings

size_t n
number of strings in the array or -1 for NULL terminated arrays

const char *str
string to match with

Description

Returns index of **str** in the **array** or -EINVAL, just like *match_string()*. Uses sysfs_streq instead of strcmp for matching.

This routine will look for a string in an array of strings up to the n-th element in the array or until the first NULL element.

Historically the value of -1 for **n**, was used to search in arrays that are NULL terminated. However, the function does not make a distinction when finishing the search: either **n** elements have been compared OR the first NULL element was found.

char ***strreplace**(char *str, char old, char new)
Replace all occurrences of character in string.

Parameters

char *str
The string to operate on.

char old
The character being replaced.

char new
The character **old** is replaced with.

Description

Replaces the each **old** character with a **new** one in the given string **str**.

Return

pointer to the string **str** itself.

void **memcpy_and_pad**(void *dest, size_t dest_len, const void *src, size_t count, int pad)
Copy one buffer to another with padding

Parameters

void *dest
Where to copy to

size_t dest_len
The destination buffer size

const void *src
Where to copy from

size_t count
The number of bytes to copy

int pad
Character to use for padding if space is left in destination.

String Manipulation

unsafe_memcpy

`unsafe_memcpy (dst, src, bytes, justification)`

memcpy implementation with no FORTIFY bounds checking

Parameters

dst
Destination memory address to write to

src
Source memory address to read from

bytes
How many bytes to write to **dst** from **src**

justification
Free-form text or comment describing why the use is needed

Description

This should be used for corner cases where the compiler cannot do the right thing, or during transitions between APIs, etc. It should be used very rarely, and includes a place for justification detailing where bounds checking has happened, and why existing solutions cannot be employed.

char ***strncpy**(char *const p, const char *q, __kernel_size_t size)
Copy a string to memory with non-guaranteed NUL padding

Parameters

char * const p
pointer to destination of copy

const char *q
pointer to NUL-terminated source string to copy

__kernel_size_t size
bytes to write at **p**

Description

If strlen(**q**) >= **size**, the copy of **q** will stop after **size** bytes, and **p** will NOT be NUL-terminated

If strlen(**q**) < **size**, following the copy of **q**, trailing NUL bytes will be written to **p** until **size** total bytes have been written.

Do not use this function. While FORTIFY_SOURCE tries to avoid over-reads of **q**, it cannot defend against writing unterminated results to **p**. Using strncpy() remains ambiguous and fragile. Instead, please choose an alternative, so that the expectation of **p**'s contents is unambiguous:

p needs to be:	padded to **size**	not padded
NUL-terminated	strscpy_pad()	strscpy()
not NUL-terminated	strtomem_pad()	strtomem()

Note strscpy*()'s differing return values for detecting truncation, and strtomem*()'s expectation that the destination is marked with __nonstring when it is a character array.

__kernel_size_t **strnlen**(const char *const p, __kernel_size_t maxlen)
Return bounded count of characters in a NUL-terminated string

Parameters

const char * const p
pointer to NUL-terminated string to count.

__kernel_size_t maxlen
maximum number of characters to count.

Description

Returns number of characters in **p** (NOT including the final NUL), or **maxlen**, if no NUL has been found up to there.

strlen

`strlen (p)`

Return count of characters in a NUL-terminated string

Parameters

p
pointer to NUL-terminated string to count.

Description

Do not use this function unless the string length is known at compile-time. When **p** is unterminated, this function may crash or return unexpected counts that could lead to memory content exposures. Prefer *strnlen()*.

Returns number of characters in **p** (NOT including the final NUL).

size_t **strlcpy**(char *const p, const char *const q, size_t size)

Copy a string into another string buffer

Parameters

char * const p
pointer to destination of copy

const char * const q
pointer to NUL-terminated source string to copy

size_t size
maximum number of bytes to write at **p**

Description

If strlen(**q**) >= **size**, the copy of **q** will be truncated at **size** - 1 bytes. **p** will always be NUL-terminated.

Do not use this function. While FORTIFY_SOURCE tries to avoid over-reads when calculating strlen(**q**), it is still possible. Prefer strscpy(), though note its different return values for detecting truncation.

Returns total number of bytes written to **p**, including terminating NUL.

ssize_t **strscpy**(char *const p, const char *const q, size_t size)

Copy a C-string into a sized buffer

Parameters

char * const p
Where to copy the string to

const char * const q
Where to copy the string from

size_t size
Size of destination buffer

Description

Copy the source string **q**, or as much of it as fits, into the destination **p** buffer. The behavior is undefined if the string buffers overlap. The destination **p** buffer is always NUL terminated, unless it's zero-sized.

Preferred to strlcpy() since the API doesn't require reading memory from the source **q** string beyond the specified **size** bytes, and since the return value is easier to error-check than strlcpy()'s. In addition, the implementation is robust to the string changing out from underneath it, unlike the current strlcpy() implementation.

Preferred to strncpy() since it always returns a valid string, and doesn't unnecessarily force the tail of the destination buffer to be zero padded. If padding is desired please use strscpy_pad().

Returns the number of characters copied in **p** (not including the trailing `NUL`) or -E2BIG if **size** is 0 or the copy of **q** was truncated.

size_t **strlcat**(char *const p, const char *const q, size_t avail)

Append a string to an existing string

Parameters

char * const p
pointer to `NUL-terminated` string to append to

const char * const q
pointer to `NUL-terminated` string to append from

size_t avail
Maximum bytes available in **p**

Description

Appends `NUL-terminated` string **q** after the `NUL-terminated` string at **p**, but will not write beyond **avail** bytes total, potentially truncating the copy from **q**. **p** will stay `NUL-terminated` only if a `NUL` already existed within the **avail** bytes of **p**. If so, the resulting number of bytes copied from **q** will be at most "**avail** - strlen(**p**) - 1".

Do not use this function. While FORTIFY_SOURCE tries to avoid read and write overflows, this is only possible when the sizes of **p** and **q** are known to the compiler. Prefer building the string with formatting, via *scnprintf()*, seq_buf, or similar.

Returns total bytes that _would_ have been contained by **p** regardless of truncation, similar to *snprintf()*. If return value is >= **avail**, the string has been truncated.

char ***strcat**(char *const p, const char *q)

Append a string to an existing string

Parameters

char * const p
pointer to NUL-terminated string to append to

const char *q
pointer to NUL-terminated source string to append from

Description

Do not use this function. While FORTIFY_SOURCE tries to avoid read and write overflows, this is only possible when the destination buffer size is known to the compiler. Prefer building the string with formatting, via *scnprintf()* or similar. At the very least, use *strncat()*.

Returns **p**.

char ***strncat**(char *const p, const char *const q, __kernel_size_t count)
    Append a string to an existing string

Parameters

char * const p
    pointer to NUL-terminated string to append to

const char * const q
    pointer to source string to append from

__kernel_size_t count
    Maximum bytes to read from **q**

Description

Appends at most **count** bytes from **q** (stopping at the first NUL byte) after the NUL-terminated string at **p**. **p** will be NUL-terminated.

Do not use this function. While FORTIFY_SOURCE tries to avoid read and write overflows, this is only possible when the sizes of **p** and **q** are known to the compiler. Prefer building the string with formatting, via *scnprintf()* or similar.

Returns **p**.

char ***strcpy**(char *const p, const char *const q)
    Copy a string into another string buffer

Parameters

char * const p
    pointer to destination of copy

const char * const q
    pointer to NUL-terminated source string to copy

Description

Do not use this function. While FORTIFY_SOURCE tries to avoid overflows, this is only possible when the sizes of **q** and **p** are known to the compiler. Prefer strscpy(), though note its different return values for detecting truncation.

Returns **p**.

int **strncasecmp**(const char *s1, const char *s2, size_t len)
    Case insensitive, length-limited string comparison

Parameters

const char *s1
    One string

const char *s2
    The other string

size_t len
    the maximum number of characters to compare

char *`stpcpy`(char *__restrict__ dest, const char *__restrict__ src)

copy a string from src to dest returning a pointer to the new end of dest, including src's `NUL-terminator`. May overrun dest.

Parameters

`char *__restrict__ dest`
pointer to end of string being copied into. Must be large enough to receive copy.

`const char *__restrict__ src`
pointer to the beginning of string being copied from. Must not overlap dest.

Description

stpcpy differs from strcpy in a key way: the return value is a pointer to the new `NUL-terminating` character in **dest**. (For strcpy, the return value is a pointer to the start of **dest**). This interface is considered unsafe as it doesn't perform bounds checking of the inputs. As such it's not recommended for usage. Instead, its definition is provided in case the compiler lowers other libcalls to stpcpy.

int `strcmp`(const char *cs, const char *ct)

Compare two strings

Parameters

`const char *cs`
One string

`const char *ct`
Another string

int `strncmp`(const char *cs, const char *ct, size_t count)

Compare two length-limited strings

Parameters

`const char *cs`
One string

`const char *ct`
Another string

`size_t count`
The maximum number of bytes to compare

char *`strchr`(const char *s, int c)

Find the first occurrence of a character in a string

Parameters

`const char *s`
The string to be searched

`int c`
The character to search for

Description

Note that the `NUL-terminator` is considered part of the string, and can be searched for.

char ***strchrnul**(const char *s, int c)

Find and return a character in a string, or end of string

Parameters

const char *s
The string to be searched

int c
The character to search for

Description

Returns pointer to first occurrence of 'c' in s. If c is not found, then return a pointer to the null byte at the end of s.

char ***strrchr**(const char *s, int c)

Find the last occurrence of a character in a string

Parameters

const char *s
The string to be searched

int c
The character to search for

char ***strnchr**(const char *s, size_t count, int c)

Find a character in a length limited string

Parameters

const char *s
The string to be searched

size_t count
The number of characters to be searched

int c
The character to search for

Description

Note that the `NUL-terminator` is considered part of the string, and can be searched for.

size_t **strspn**(const char *s, const char *accept)

Calculate the length of the initial substring of **s** which only contain letters in **accept**

Parameters

const char *s
The string to be searched

const char *accept
The string to search for

size_t **strcspn**(const char *s, const char *reject)

Calculate the length of the initial substring of **s** which does not contain letters in **reject**

Parameters

const char *s
The string to be searched

const char *reject
The string to avoid

char ***strpbrk**(const char *cs, const char *ct)
Find the first occurrence of a set of characters

Parameters

const char *cs
The string to be searched

const char *ct
The characters to search for

char ***strsep**(char **s, const char *ct)
Split a string into tokens

Parameters

char **s
The string to be searched

const char *ct
The characters to search for

Description

strsep() updates **s** to point after the token, ready for the next call.

It returns empty tokens, too, behaving exactly like the libc function of that name. In fact, it was stolen from glibc2 and de-fancy-fied. Same semantics, slimmer shape. ;)

void ***memset**(void *s, int c, size_t count)
Fill a region of memory with the given value

Parameters

void *s
Pointer to the start of the area.

int c
The byte to fill the area with

size_t count
The size of the area.

Description

Do not use memset() to access IO space, use memset_io() instead.

void ***memset16**(uint16_t *s, uint16_t v, size_t count)
Fill a memory area with a uint16_t

Parameters

uint16_t *s
Pointer to the start of the area.

uint16_t v
The value to fill the area with

size_t count
The number of values to store

Description

Differs from memset() in that it fills with a uint16_t instead of a byte. Remember that **count** is the number of uint16_ts to store, not the number of bytes.

void ***memset32**(uint32_t *s, uint32_t v, size_t count)
Fill a memory area with a uint32_t

Parameters

uint32_t *s
Pointer to the start of the area.

uint32_t v
The value to fill the area with

size_t count
The number of values to store

Description

Differs from memset() in that it fills with a uint32_t instead of a byte. Remember that **count** is the number of uint32_ts to store, not the number of bytes.

void ***memset64**(uint64_t *s, uint64_t v, size_t count)
Fill a memory area with a uint64_t

Parameters

uint64_t *s
Pointer to the start of the area.

uint64_t v
The value to fill the area with

size_t count
The number of values to store

Description

Differs from memset() in that it fills with a uint64_t instead of a byte. Remember that **count** is the number of uint64_ts to store, not the number of bytes.

void ***memcpy**(void *dest, const void *src, size_t count)
Copy one area of memory to another

Parameters

void *dest
Where to copy to

const void *src
Where to copy from

size_t count
The size of the area.

Description

You should not use this function to access IO space, use memcpy_toio() or memcpy_fromio() instead.

void ***memmove**(void *dest, const void *src, size_t count)

Copy one area of memory to another

Parameters

void *dest
Where to copy to

const void *src
Where to copy from

size_t count
The size of the area.

Description

Unlike memcpy(), *memmove()* copes with overlapping areas.

__visible int **memcmp**(const void *cs, const void *ct, size_t count)

Compare two areas of memory

Parameters

const void *cs
One area of memory

const void *ct
Another area of memory

size_t count
The size of the area.

int **bcmp**(const void *a, const void *b, size_t len)

returns 0 if and only if the buffers have identical contents.

Parameters

const void *a
pointer to first buffer.

const void *b
pointer to second buffer.

size_t len
size of buffers.

Description

The sign or magnitude of a non-zero return value has no particular meaning, and architectures may implement their own more efficient *bcmp()*. So while this particular implementation is a simple (tail) call to memcmp, do not rely on anything but whether the return value is zero or non-zero.

void ***memscan**(void *addr, int c, size_t size)

Find a character in an area of memory.

Parameters

void *addr
    The memory area

int c
    The byte to search for

size_t size
    The size of the area.

Description

returns the address of the first occurrence of **c**, or 1 byte past the area if **c** is not found

char ***strstr**(const char *s1, const char *s2)
    Find the first substring in a NUL terminated string

Parameters

const char *s1
    The string to be searched

const char *s2
    The string to search for

char ***strnstr**(const char *s1, const char *s2, size_t len)
    Find the first substring in a length-limited string

Parameters

const char *s1
    The string to be searched

const char *s2
    The string to search for

size_t len
    the maximum number of characters to search

void ***memchr**(const void *s, int c, size_t n)
    Find a character in an area of memory.

Parameters

const void *s
    The memory area

int c
    The byte to search for

size_t n
    The size of the area.

Description

returns the address of the first occurrence of **c**, or NULL if **c** is not found

void ***memchr_inv**(const void *start, int c, size_t bytes)
    Find an unmatching character in an area of memory.

Parameters

const void *start
: The memory area

int c
: Find a character other than c

size_t bytes
: The size of the area.

Description

returns the address of the first character other than **c**, or `NULL` if the whole buffer contains just **c**.

void ***memdup_array_user**(const void __user *src, size_t n, size_t size)
: duplicate array from user space

Parameters

const void __user *src
: source address in user space

size_t n
: number of array members to copy

size_t size
: size of one array member

Return

an ERR_PTR() on failure. Result is physically contiguous, to be freed by kfree().

void ***vmemdup_array_user**(const void __user *src, size_t n, size_t size)
: duplicate array from user space

Parameters

const void __user *src
: source address in user space

size_t n
: number of array members to copy

size_t size
: size of one array member

Return

an ERR_PTR() on failure. Result may be not physically contiguous. Use kvfree() to free.

sysfs_match_string

`sysfs_match_string (_a, _s)`

: matches given string in an array

Parameters

_a
: array of strings

_s
: string to match with

Description

Helper for *__sysfs_match_string()*. Calculates the size of **a** automatically.

bool **strstarts**(const char *str, const char *prefix)

does **str** start with **prefix**?

Parameters

const char *str
string to examine

const char *prefix
prefix to look for.

void **memzero_explicit**(void *s, size_t count)

Fill a region of memory (e.g. sensitive keying data) with 0s.

Parameters

void *s
Pointer to the start of the area.

size_t count
The size of the area.

Note

usually using memset() is just fine (!), but in cases where clearing out _local_ data at the end of a scope is necessary, *memzero_explicit()* should be used instead in order to prevent the compiler from optimising away zeroing.

Description

memzero_explicit() doesn't need an arch-specific version as it just invokes the one of memset() implicitly.

const char ***kbasename**(const char *path)

return the last part of a pathname.

Parameters

const char *path
path to extract the filename from.

strtomem_pad

`strtomem_pad (dest, src, pad)`

Copy NUL-terminated string to non-NUL-terminated buffer

Parameters

dest
Pointer of destination character array (marked as __nonstring)

src
Pointer to NUL-terminated string

pad
Padding character to fill any remaining bytes of **dest** after copy

Description

This is a replacement for strncpy() uses where the destination is not a NUL-terminated string, but with bounds checking on the source size, and an explicit padding character. If padding is not required, use strtomem().

Note that the size of **dest** is not an argument, as the length of **dest** must be discoverable by the compiler.

strtomem

`strtomem (dest, src)`

Copy NUL-terminated string to non-NUL-terminated buffer

Parameters

dest
Pointer of destination character array (marked as __nonstring)

src
Pointer to NUL-terminated string

Description

This is a replacement for strncpy() uses where the destination is not a NUL-terminated string, but with bounds checking on the source size, and without trailing padding. If padding is required, use strtomem_pad().

Note that the size of **dest** is not an argument, as the length of **dest** must be discoverable by the compiler.

memset_after

`memset_after (obj, v, member)`

Set a value after a struct member to the end of a struct

Parameters

obj
Address of target struct instance

v
Byte value to repeatedly write

member
after which struct member to start writing bytes

Description

This is good for clearing padding following the given member.

memset_startat

`memset_startat (obj, v, member)`

Set a value starting at a member to the end of a struct

Parameters

obj
Address of target struct instance

v
Byte value to repeatedly write

member
struct member to start writing at

Description

Note that if there is padding between the prior member and the target member, *memset_after()* should be used to clear the prior padding.

size_t **str_has_prefix**(const char *str, const char *prefix)
Test if a string has a given prefix

Parameters

const char *str
The string to test

const char *prefix
The string to see if **str** starts with

Description

A common way to test a prefix of a string is to do:
strncmp(str, prefix, sizeof(prefix) - 1)

But this can lead to bugs due to typos, or if prefix is a pointer and not a constant. Instead use *str_has_prefix()*.

Return

- strlen(**prefix**) if **str** starts with **prefix**
- 0 if **str** does not start with **prefix**

char ***kstrdup**(const char *s, gfp_t gfp)
allocate space for and copy an existing string

Parameters

const char *s
the string to duplicate

gfp_t gfp
the GFP mask used in the kmalloc() call when allocating memory

Return

newly allocated copy of **s** or `NULL` in case of error

const char ***kstrdup_const**(const char *s, gfp_t gfp)
conditionally duplicate an existing const string

Parameters

const char *s
the string to duplicate

gfp_t gfp
the GFP mask used in the kmalloc() call when allocating memory

Note

Strings allocated by kstrdup_const should be freed by kfree_const and must not be passed to krealloc().

Return

source string if it is in .rodata section otherwise fallback to kstrdup.

char ***kstrndup**(const char *s, size_t max, gfp_t gfp)

allocate space for and copy an existing string

Parameters

const char *s
the string to duplicate

size_t max
read at most **max** chars from **s**

gfp_t gfp
the GFP mask used in the kmalloc() call when allocating memory

Note

Use *kmemdup_nul()* instead if the size is known exactly.

Return

newly allocated copy of **s** or `NULL` in case of error

void ***kmemdup**(const void *src, size_t len, gfp_t gfp)

duplicate region of memory

Parameters

const void *src
memory region to duplicate

size_t len
memory region length

gfp_t gfp
GFP mask to use

Return

newly allocated copy of **src** or `NULL` in case of error, result is physically contiguous. Use kfree() to free.

char ***kmemdup_nul**(const char *s, size_t len, gfp_t gfp)

Create a NUL-terminated string from unterminated data

Parameters

const char *s
The data to stringify

size_t len
The size of the data

gfp_t gfp
the GFP mask used in the kmalloc() call when allocating memory

Return

newly allocated copy of **s** with NUL-termination or `NULL` in case of error

void ***memdup_user**(const void __user *src, size_t len)
duplicate memory region from user space

Parameters

const void __user *src
source address in user space

size_t len
number of bytes to copy

Return

an ERR_PTR() on failure. Result is physically contiguous, to be freed by kfree().

void ***vmemdup_user**(const void __user *src, size_t len)
duplicate memory region from user space

Parameters

const void __user *src
source address in user space

size_t len
number of bytes to copy

Return

an ERR_PTR() on failure. Result may be not physically contiguous. Use kvfree() to free.

char ***strndup_user**(const char __user *s, long n)
duplicate an existing string from user space

Parameters

const char __user *s
The string to duplicate

long n
Maximum number of bytes to copy, including the trailing NUL.

Return

newly allocated copy of **s** or an ERR_PTR() in case of error

void ***memdup_user_nul**(const void __user *src, size_t len)
duplicate memory region from user space and NUL-terminate

Parameters

const void __user *src
source address in user space

size_t len
number of bytes to copy

Return

an ERR_PTR() on failure.

1.1.3 Basic Kernel Library Functions

The Linux kernel provides more basic utility functions.

Bit Operations

void **set_bit**(long nr, volatile unsigned long *addr)
Atomically set a bit in memory

Parameters

long nr
the bit to set

volatile unsigned long *addr
the address to start counting from

Description

This is a relaxed atomic operation (no implied memory barriers).

Note that **nr** may be almost arbitrarily large; this function is not restricted to acting on a single-word quantity.

void **clear_bit**(long nr, volatile unsigned long *addr)
Clears a bit in memory

Parameters

long nr
Bit to clear

volatile unsigned long *addr
Address to start counting from

Description

This is a relaxed atomic operation (no implied memory barriers).

void **change_bit**(long nr, volatile unsigned long *addr)
Toggle a bit in memory

Parameters

long nr
Bit to change

volatile unsigned long *addr
Address to start counting from

Description

This is a relaxed atomic operation (no implied memory barriers).

Note that **nr** may be almost arbitrarily large; this function is not restricted to acting on a single-word quantity.

bool **test_and_set_bit**(long nr, volatile unsigned long *addr)
Set a bit and return its old value

Parameters

long nr
Bit to set

volatile unsigned long *addr
Address to count from

Description

This is an atomic fully-ordered operation (implied full memory barrier).

bool **test_and_clear_bit**(long nr, volatile unsigned long *addr)
Clear a bit and return its old value

Parameters

long nr
Bit to clear

volatile unsigned long *addr
Address to count from

Description

This is an atomic fully-ordered operation (implied full memory barrier).

bool **test_and_change_bit**(long nr, volatile unsigned long *addr)
Change a bit and return its old value

Parameters

long nr
Bit to change

volatile unsigned long *addr
Address to count from

Description

This is an atomic fully-ordered operation (implied full memory barrier).

void **___set_bit**(unsigned long nr, volatile unsigned long *addr)
Set a bit in memory

Parameters

unsigned long nr
the bit to set

volatile unsigned long *addr
the address to start counting from

Description

Unlike set_bit(), this function is non-atomic. If it is called on the same region of memory concurrently, the effect may be that only one operation succeeds.

void **___clear_bit**(unsigned long nr, volatile unsigned long *addr)
Clears a bit in memory

Parameters

unsigned long nr
the bit to clear

volatile unsigned long *addr
the address to start counting from

Description

Unlike clear_bit(), this function is non-atomic. If it is called on the same region of memory concurrently, the effect may be that only one operation succeeds.

void **___change_bit**(unsigned long nr, volatile unsigned long *addr)
Toggle a bit in memory

Parameters

unsigned long nr
the bit to change

volatile unsigned long *addr
the address to start counting from

Description

Unlike *change_bit()*, this function is non-atomic. If it is called on the same region of memory concurrently, the effect may be that only one operation succeeds.

bool **___test_and_set_bit**(unsigned long nr, volatile unsigned long *addr)
Set a bit and return its old value

Parameters

unsigned long nr
Bit to set

volatile unsigned long *addr
Address to count from

Description

This operation is non-atomic. If two instances of this operation race, one can appear to succeed but actually fail.

bool **___test_and_clear_bit**(unsigned long nr, volatile unsigned long *addr)
Clear a bit and return its old value

Parameters

unsigned long nr
Bit to clear

volatile unsigned long *addr
Address to count from

Description

This operation is non-atomic. If two instances of this operation race, one can appear to succeed but actually fail.

bool **`___test_and_change_bit`**(unsigned long nr, volatile unsigned long *addr)

Change a bit and return its old value

Parameters

`unsigned long nr`
Bit to change

`volatile unsigned long *addr`
Address to count from

Description

This operation is non-atomic. If two instances of this operation race, one can appear to succeed but actually fail.

bool **`_test_bit`**(unsigned long nr, volatile const unsigned long *addr)

Determine whether a bit is set

Parameters

`unsigned long nr`
bit number to test

`const volatile unsigned long *addr`
Address to start counting from

bool **`_test_bit_acquire`**(unsigned long nr, volatile const unsigned long *addr)

Determine, with acquire semantics, whether a bit is set

Parameters

`unsigned long nr`
bit number to test

`const volatile unsigned long *addr`
Address to start counting from

void **`clear_bit_unlock`**(long nr, volatile unsigned long *addr)

Clear a bit in memory, for unlock

Parameters

`long nr`
the bit to set

`volatile unsigned long *addr`
the address to start counting from

Description

This operation is atomic and provides release barrier semantics.

void **`__clear_bit_unlock`**(long nr, volatile unsigned long *addr)

Clears a bit in memory

Parameters

long nr
Bit to clear

volatile unsigned long *addr
Address to start counting from

Description

This is a non-atomic operation but implies a release barrier before the memory operation. It can be used for an unlock if no other CPUs can concurrently modify other bits in the word.

bool **test_and_set_bit_lock**(long nr, volatile unsigned long *addr)
Set a bit and return its old value, for lock

Parameters

long nr
Bit to set

volatile unsigned long *addr
Address to count from

Description

This operation is atomic and provides acquire barrier semantics if the returned value is 0. It can be used to implement bit locks.

bool **clear_bit_unlock_is_negative_byte**(long nr, volatile unsigned long *addr)
Clear a bit in memory and test if bottom byte is negative, for unlock.

Parameters

long nr
the bit to clear

volatile unsigned long *addr
the address to start counting from

Description

This operation is atomic and provides release barrier semantics.

This is a bit of a one-trick-pony for the filemap code, which clears PG_locked and tests PG_waiters,

Bitmap Operations

bitmaps provide an array of bits, implemented using an array of unsigned longs. The number of valid bits in a given bitmap does _not_ need to be an exact multiple of BITS_PER_LONG.

The possible unused bits in the last, partially used word of a bitmap are 'don't care'. The implementation makes no particular effort to keep them zero. It ensures that their value will not affect the results of any operation. The bitmap operations that return Boolean (bitmap_empty, for example) or scalar (bitmap_weight, for example) results carefully filter out these unused bits from impacting their results.

The byte ordering of bitmaps is more natural on little endian architectures. See the big-endian headers include/asm-ppc64/bitops.h and include/asm-s390/bitops.h for the best explanations of this ordering.

The DECLARE_BITMAP(name,bits) macro, in linux/types.h, can be used to declare an array named 'name' of just enough unsigned longs to contain all bit positions from 0 to 'bits' - 1.

The available bitmap operations and their rough meaning in the case that the bitmap is a single unsigned long are thus:

The generated code is more efficient when nbits is known at compile-time and at most BITS_PER_LONG.

```
bitmap_zero(dst, nbits)                     *dst = 0UL
bitmap_fill(dst, nbits)                     *dst = ~0UL
bitmap_copy(dst, src, nbits)                *dst = *src
bitmap_and(dst, src1, src2, nbits)          *dst = *src1 & *src2
bitmap_or(dst, src1, src2, nbits)           *dst = *src1 | *src2
bitmap_xor(dst, src1, src2, nbits)          *dst = *src1 ^ *src2
bitmap_andnot(dst, src1, src2, nbits)       *dst = *src1 & ~(*src2)
bitmap_complement(dst, src, nbits)          *dst = ~(*src)
bitmap_equal(src1, src2, nbits)             Are *src1 and *src2 equal?
bitmap_intersects(src1, src2, nbits)        Do *src1 and *src2 overlap?
bitmap_subset(src1, src2, nbits)            Is *src1 a subset of *src2?
bitmap_empty(src, nbits)                    Are all bits zero in *src?
bitmap_full(src, nbits)                     Are all bits set in *src?
bitmap_weight(src, nbits)                   Hamming Weight: number set bits
bitmap_weight_and(src1, src2, nbits)        Hamming Weight of and'ed bitmap
bitmap_set(dst, pos, nbits)                 Set specified bit area
bitmap_clear(dst, pos, nbits)               Clear specified bit area
bitmap_find_next_zero_area(buf, len, pos, n, mask)  Find bit free area
bitmap_find_next_zero_area_off(buf, len, pos, n, mask, mask_off)  as above
bitmap_shift_right(dst, src, n, nbits)      *dst = *src >> n
bitmap_shift_left(dst, src, n, nbits)       *dst = *src << n
bitmap_cut(dst, src, first, n, nbits)       Cut n bits from first, copy rest
bitmap_replace(dst, old, new, mask, nbits)  *dst = (*old & ~(*mask)) | (*new &
 *mask)
bitmap_remap(dst, src, old, new, nbits)     *dst = map(old, new)(src)
bitmap_bitremap(oldbit, old, new, nbits)    newbit = map(old, new)(oldbit)
bitmap_onto(dst, orig, relmap, nbits)       *dst = orig relative to relmap
bitmap_fold(dst, orig, sz, nbits)           dst bits = orig bits mod sz
bitmap_parse(buf, buflen, dst, nbits)       Parse bitmap dst from kernel buf
bitmap_parse_user(ubuf, ulen, dst, nbits)   Parse bitmap dst from user buf
bitmap_parselist(buf, dst, nbits)           Parse bitmap dst from kernel buf
bitmap_parselist_user(buf, dst, nbits)      Parse bitmap dst from user buf
bitmap_find_free_region(bitmap, bits, order)  Find and allocate bit region
bitmap_release_region(bitmap, pos, order)   Free specified bit region
bitmap_allocate_region(bitmap, pos, order)  Allocate specified bit region
bitmap_from_arr32(dst, buf, nbits)          Copy nbits from u32[] buf to dst
bitmap_from_arr64(dst, buf, nbits)          Copy nbits from u64[] buf to dst
bitmap_to_arr32(buf, src, nbits)            Copy nbits from buf to u32[] dst
bitmap_to_arr64(buf, src, nbits)            Copy nbits from buf to u64[] dst
bitmap_get_value8(map, start)               Get 8bit value from map at start
bitmap_set_value8(map, value, start)        Set 8bit value to map at start
```

Note, bitmap_zero() and bitmap_fill() operate over the region of unsigned longs, that is, bits behind bitmap till the unsigned long boundary will be zeroed or filled as well. Consider to use bitmap_clear() or bitmap_set() to make explicit zeroing or filling respectively.

Also the following operations in asm/bitops.h apply to bitmaps.:

```
set_bit(bit, addr)                  *addr |= bit
clear_bit(bit, addr)                *addr &= ~bit
change_bit(bit, addr)               *addr ^= bit
test_bit(bit, addr)                 Is bit set in *addr?
test_and_set_bit(bit, addr)         Set bit and return old value
test_and_clear_bit(bit, addr)       Clear bit and return old value
test_and_change_bit(bit, addr)      Change bit and return old value
find_first_zero_bit(addr, nbits)    Position first zero bit in *addr
find_first_bit(addr, nbits)         Position first set bit in *addr
find_next_zero_bit(addr, nbits, bit)
                                    Position next zero bit in *addr >= bit
find_next_bit(addr, nbits, bit)     Position next set bit in *addr >= bit
find_next_and_bit(addr1, addr2, nbits, bit)
                                    Same as find_next_bit, but in
                                    (*addr1 & *addr2)
```

void **__bitmap_shift_right**(unsigned long *dst, const unsigned long *src, unsigned shift, unsigned nbits)

logical right shift of the bits in a bitmap

Parameters

unsigned long *dst
destination bitmap

const unsigned long *src
source bitmap

unsigned shift
shift by this many bits

unsigned nbits
bitmap size, in bits

Description

Shifting right (dividing) means moving bits in the MS -> LS bit direction. Zeros are fed into the vacated MS positions and the LS bits shifted off the bottom are lost.

void **__bitmap_shift_left**(unsigned long *dst, const unsigned long *src, unsigned int shift, unsigned int nbits)

logical left shift of the bits in a bitmap

Parameters

unsigned long *dst
destination bitmap

const unsigned long *src
source bitmap

unsigned int shift
shift by this many bits

unsigned int nbits
bitmap size, in bits

Description

Shifting left (multiplying) means moving bits in the LS -> MS direction. Zeros are fed into the vacated LS bit positions and those MS bits shifted off the top are lost.

void **bitmap_cut**(unsigned long *dst, const unsigned long *src, unsigned int first, unsigned int cut, unsigned int nbits)

remove bit region from bitmap and right shift remaining bits

Parameters

unsigned long *dst
destination bitmap, might overlap with src

const unsigned long *src
source bitmap

unsigned int first
start bit of region to be removed

unsigned int cut
number of bits to remove

unsigned int nbits
bitmap size, in bits

Description

Set the n-th bit of **dst** iff the n-th bit of **src** is set and n is less than **first**, or the m-th bit of **src** is set for any m such that **first** <= n < nbits, and m = n + **cut**.

In pictures, example for a big-endian 32-bit architecture:

The **src** bitmap is:

```
31                                   63
|                                    |
10000000 11000001 11110010 00010101  10000000 11000001 01110010 00010101
                |  |              |                                    |
               16  14             0                                   32
```

if **cut** is 3, and **first** is 14, bits 14-16 in **src** are cut and **dst** is:

```
31                                   63
|                                    |
10110000 00011000 00110010 00010101  00010000 00011000 00101110 01000010
                   |              |                                    |
                   14 (bit 17     0                                   32
                       from @src)
```

Note that **dst** and **src** might overlap partially or entirely.

This is implemented in the obvious way, with a shift and carry step for each moved bit. Optimisation is left as an exercise for the compiler.

unsigned long **bitmap_find_next_zero_area_off**(unsigned long *map, unsigned long size, unsigned long start, unsigned int nr, unsigned long align_mask, unsigned long align_offset)

find a contiguous aligned zero area

Parameters

unsigned long *map
The address to base the search on

unsigned long size
The bitmap size in bits

unsigned long start
The bitnumber to start searching at

unsigned int nr
The number of zeroed bits we're looking for

unsigned long align_mask
Alignment mask for zero area

unsigned long align_offset
Alignment offset for zero area.

Description

The **align_mask** should be one less than a power of 2; the effect is that the bit offset of all zero areas this function finds plus **align_offset** is multiple of that power of 2.

int **bitmap_parse_user**(const char __user *ubuf, unsigned int ulen, unsigned long *maskp, int nmaskbits)

convert an ASCII hex string in a user buffer into a bitmap

Parameters

const char __user *ubuf
pointer to user buffer containing string.

unsigned int ulen
buffer size in bytes. If string is smaller than this then it must be terminated with a 0.

unsigned long *maskp
pointer to bitmap array that will contain result.

int nmaskbits
size of bitmap, in bits.

int **bitmap_print_to_pagebuf**(bool list, char *buf, const unsigned long *maskp, int nmaskbits)

convert bitmap to list or hex format ASCII string

Parameters

bool list
indicates whether the bitmap must be list

char *buf
page aligned buffer into which string is placed

const unsigned long *maskp
pointer to bitmap to convert

int nmaskbits
size of bitmap, in bits

Description

Output format is a comma-separated list of decimal numbers and ranges if list is specified or hex digits grouped into comma-separated sets of 8 digits/set. Returns the number of characters written to buf.

It is assumed that **buf** is a pointer into a PAGE_SIZE, page-aligned area and that sufficient storage remains at **buf** to accommodate the *bitmap_print_to_pagebuf()* output. Returns the number of characters actually printed to **buf**, excluding terminating '0'.

int **bitmap_print_bitmask_to_buf**(char *buf, const unsigned long *maskp, int nmaskbits, loff_t off, size_t count)

convert bitmap to hex bitmask format ASCII string

Parameters

char *buf
buffer into which string is placed

const unsigned long *maskp
pointer to bitmap to convert

int nmaskbits
size of bitmap, in bits

loff_t off
in the string from which we are copying, We copy to **buf**

size_t count
the maximum number of bytes to print

Description

The *bitmap_print_to_pagebuf()* is used indirectly via its cpumap wrapper cpumap_print_to_pagebuf() or directly by drivers to export hexadecimal bitmask and decimal list to userspace by sysfs ABI. Drivers might be using a normal attribute for this kind of ABIs. A normal attribute typically has show entry as below:

```
static ssize_t example_attribute_show(struct device *dev,
            struct device_attribute *attr, char *buf)
{
    ...
    return bitmap_print_to_pagebuf(true, buf, &mask, nr_trig_max);
}
```

show entry of attribute has no offset and count parameters and this means the file is limited to one page only. *bitmap_print_to_pagebuf()* API works terribly well for this kind of normal attribute with buf parameter and without offset, count:

```
bitmap_print_to_pagebuf(bool list, char *buf, const unsigned long *maskp,
                        int nmaskbits)
```

```
{
}
```

The problem is once we have a large bitmap, we have a chance to get a bitmask or list more than one page. Especially for list, it could be as complex as 0,3,5,7,9,… We have no simple way to know it exact size. It turns out bin_attribute is a way to break this limit. bin_attribute has show entry as below:

```
static ssize_t
example_bin_attribute_show(struct file *filp, struct kobject *kobj,
            struct bin_attribute *attr, char *buf,
            loff_t offset, size_t count)
{
   ...
}
```

With the new offset and count parameters, this makes sysfs ABI be able to support file size more than one page. For example, offset could be >= 4096. *bitmap_print_bitmask_to_buf()*, *bitmap_print_list_to_buf()* wit their cpumap wrapper cpumap_print_bitmask_to_buf(), cpumap_print_list_to_buf() make those drivers be able to support large bitmask and list after they move to use bin_attribute. In result, we have to pass the corresponding parameters such as off, count from bin_attribute show entry to this API.

The role of cpumap_print_bitmask_to_buf() and cpumap_print_list_to_buf() is similar with cpumap_print_to_pagebuf(), the difference is that *bitmap_print_to_pagebuf()* mainly serves sysfs attribute with the assumption the destination buffer is exactly one page and won't be more than one page. cpumap_print_bitmask_to_buf() and cpumap_print_list_to_buf(), on the other hand, mainly serves bin_attribute which doesn't work with exact one page, and it can break the size limit of converted decimal list and hexadecimal bitmask.

WARNING!

This function is not a replacement for *sprintf()* or *bitmap_print_to_pagebuf()*. It is intended to workaround sysfs limitations discussed above and should be used carefully in general case for the following reasons:

- Time complexity is O(nbits^2/count), comparing to O(nbits) for *snprintf()*.
- Memory complexity is O(nbits), comparing to O(1) for *snprintf()*.
- **off** and **count** are NOT offset and number of bits to print.
- If printing part of bitmap as list, the resulting string is not a correct list representation of bitmap. Particularly, some bits within or out of related interval may be erroneously set or unset. The format of the string may be broken, so bitmap_parselist-like parser may fail parsing it.
- If printing the whole bitmap as list by parts, user must ensure the order of calls of the function such that the offset is incremented linearly.
- If printing the whole bitmap as list by parts, user must keep bitmap unchanged between the very first and very last call. Otherwise concatenated result may be incorrect, and format may be broken.

Returns the number of characters actually printed to **buf** int **bitmap_print_list_to_buf**(char *buf, const unsigned long *maskp, int nmaskbits, loff_t off, size_t count)

convert bitmap to decimal list format ASCII string

Parameters

char *buf
buffer into which string is placed

const unsigned long *maskp
pointer to bitmap to convert

int nmaskbits
size of bitmap, in bits

loff_t off
in the string from which we are copying, We copy to **buf**

size_t count
the maximum number of bytes to print

Description

Everything is same with the above *bitmap_print_bitmask_to_buf()* except the print format.

int **bitmap_parselist**(const char *buf, unsigned long *maskp, int nmaskbits)

convert list format ASCII string to bitmap

Parameters

const char *buf
read user string from this buffer; must be terminated with a 0 or n.

unsigned long *maskp
write resulting mask here

int nmaskbits
number of bits in mask to be written

Description

Input format is a comma-separated list of decimal numbers and ranges. Consecutively set bits are shown as two hyphen-separated decimal numbers, the smallest and largest bit numbers set in the range. Optionally each range can be postfixed to denote that only parts of it should be set. The range will divided to groups of specific size. From each group will be used only defined amount of bits. Syntax: range:used_size/group_size

Example

0-1023:2/256 ==> 0,1,256,257,512,513,768,769 The value 'N' can be used as a dynamically substituted token for the maximum allowed value; i.e (nmaskbits - 1). Keep in mind that it is dynamic, so if system changes cause the bitmap width to change, such as more cores in a CPU list, then any ranges using N will also change.

Return

0 on success, -errno on invalid input strings. Error values:

- `-EINVAL`: wrong region format
- `-EINVAL`: invalid character in string

- `-ERANGE`: bit number specified too large for mask
- `-EOVERFLOW`: integer overflow in the input parameters

int **bitmap_parselist_user**(const char __user *ubuf, unsigned int ulen, unsigned long *maskp, int nmaskbits)

convert user buffer's list format ASCII string to bitmap

Parameters

const char __user *ubuf
pointer to user buffer containing string.

unsigned int ulen
buffer size in bytes. If string is smaller than this then it must be terminated with a 0.

unsigned long *maskp
pointer to bitmap array that will contain result.

int nmaskbits
size of bitmap, in bits.

Description

Wrapper for bitmap_parselist(), providing it with user buffer.

int **bitmap_parse**(const char *start, unsigned int buflen, unsigned long *maskp, int nmaskbits)

convert an ASCII hex string into a bitmap.

Parameters

const char *start
pointer to buffer containing string.

unsigned int buflen
buffer size in bytes. If string is smaller than this then it must be terminated with a 0 or n. In that case, UINT_MAX may be provided instead of string length.

unsigned long *maskp
pointer to bitmap array that will contain result.

int nmaskbits
size of bitmap, in bits.

Description

Commas group hex digits into chunks. Each chunk defines exactly 32 bits of the resultant bitmask. No chunk may specify a value larger than 32 bits (`-EOVERFLOW`), and if a chunk specifies a smaller value then leading 0-bits are prepended. `-EINVAL` is returned for illegal characters. Grouping such as "1,,5", ",44", "," or "" is allowed. Leading, embedded and trailing whitespace accepted.

void **bitmap_remap**(unsigned long *dst, const unsigned long *src, const unsigned long *old, const unsigned long *new, unsigned int nbits)

Apply map defined by a pair of bitmaps to another bitmap

Parameters

unsigned long *dst
remapped result

const unsigned long *src
subset to be remapped

const unsigned long *old
defines domain of map

const unsigned long *new
defines range of map

unsigned int nbits
number of bits in each of these bitmaps

Description

Let **old** and **new** define a mapping of bit positions, such that whatever position is held by the n-th set bit in **old** is mapped to the n-th set bit in **new**. In the more general case, allowing for the possibility that the weight 'w' of **new** is less than the weight of **old**, map the position of the n-th set bit in **old** to the position of the m-th set bit in **new**, where m == n % w.

If either of the **old** and **new** bitmaps are empty, or if **src** and **dst** point to the same location, then this routine copies **src** to **dst**.

The positions of unset bits in **old** are mapped to themselves (the identify map).

Apply the above specified mapping to **src**, placing the result in **dst**, clearing any bits previously set in **dst**.

For example, lets say that **old** has bits 4 through 7 set, and **new** has bits 12 through 15 set. This defines the mapping of bit position 4 to 12, 5 to 13, 6 to 14 and 7 to 15, and of all other bit positions unchanged. So if say **src** comes into this routine with bits 1, 5 and 7 set, then **dst** should leave with bits 1, 13 and 15 set.

int **bitmap_bitremap**(int oldbit, const unsigned long *old, const unsigned long *new, int bits)
Apply map defined by a pair of bitmaps to a single bit

Parameters

int oldbit
bit position to be mapped

const unsigned long *old
defines domain of map

const unsigned long *new
defines range of map

int bits
number of bits in each of these bitmaps

Description

Let **old** and **new** define a mapping of bit positions, such that whatever position is held by the n-th set bit in **old** is mapped to the n-th set bit in **new**. In the more general case, allowing for the possibility that the weight 'w' of **new** is less than the weight of **old**, map the position of the n-th set bit in **old** to the position of the m-th set bit in **new**, where m == n % w.

The positions of unset bits in **old** are mapped to themselves (the identify map).

Apply the above specified mapping to bit position **oldbit**, returning the new bit position.

For example, lets say that **old** has bits 4 through 7 set, and **new** has bits 12 through 15 set. This defines the mapping of bit position 4 to 12, 5 to 13, 6 to 14 and 7 to 15, and of all other bit positions unchanged. So if say **oldbit** is 5, then this routine returns 13.

int **bitmap_find_free_region**(unsigned long *bitmap, unsigned int bits, int order)

find a contiguous aligned mem region

Parameters

unsigned long *bitmap
array of unsigned longs corresponding to the bitmap

unsigned int bits
number of bits in the bitmap

int order
region size (log base 2 of number of bits) to find

Description

Find a region of free (zero) bits in a **bitmap** of **bits** bits and allocate them (set them to one). Only consider regions of length a power (**order**) of two, aligned to that power of two, which makes the search algorithm much faster.

Return the bit offset in bitmap of the allocated region, or -errno on failure.

void **bitmap_release_region**(unsigned long *bitmap, unsigned int pos, int order)

release allocated bitmap region

Parameters

unsigned long *bitmap
array of unsigned longs corresponding to the bitmap

unsigned int pos
beginning of bit region to release

int order
region size (log base 2 of number of bits) to release

Description

This is the complement to __bitmap_find_free_region() and releases the found region (by clearing it in the bitmap).

No return value.

int **bitmap_allocate_region**(unsigned long *bitmap, unsigned int pos, int order)

allocate bitmap region

Parameters

unsigned long *bitmap
array of unsigned longs corresponding to the bitmap

unsigned int pos
beginning of bit region to allocate

int order
region size (log base 2 of number of bits) to allocate

Description

Allocate (set bits in) a specified region of a bitmap.

Return 0 on success, or -EBUSY if specified region wasn't free (not all bits were zero).

void **bitmap_copy_le**(unsigned long *dst, const unsigned long *src, unsigned int nbits)
copy a bitmap, putting the bits into little-endian order.

Parameters

unsigned long *dst
destination buffer

const unsigned long *src
bitmap to copy

unsigned int nbits
number of bits in the bitmap

Description

Require nbits % BITS_PER_LONG == 0.

void **bitmap_from_arr32**(unsigned long *bitmap, const u32 *buf, unsigned int nbits)
copy the contents of u32 array of bits to bitmap

Parameters

unsigned long *bitmap
array of unsigned longs, the destination bitmap

const u32 *buf
array of u32 (in host byte order), the source bitmap

unsigned int nbits
number of bits in **bitmap**

void **bitmap_to_arr32**(u32 *buf, const unsigned long *bitmap, unsigned int nbits)
copy the contents of bitmap to a u32 array of bits

Parameters

u32 *buf
array of u32 (in host byte order), the dest bitmap

const unsigned long *bitmap
array of unsigned longs, the source bitmap

unsigned int nbits
number of bits in **bitmap**

void **bitmap_from_arr64**(unsigned long *bitmap, const u64 *buf, unsigned int nbits)
copy the contents of u64 array of bits to bitmap

Parameters

unsigned long *bitmap
array of unsigned longs, the destination bitmap

const u64 *buf
array of u64 (in host byte order), the source bitmap

unsigned int nbits
number of bits in **bitmap**

void **bitmap_to_arr64**(u64 *buf, const unsigned long *bitmap, unsigned int nbits)
copy the contents of bitmap to a u64 array of bits

Parameters

u64 *buf
array of u64 (in host byte order), the dest bitmap

const unsigned long *bitmap
array of unsigned longs, the source bitmap

unsigned int nbits
number of bits in **bitmap**

int **bitmap_print_to_buf**(bool list, char *buf, const unsigned long *maskp, int nmaskbits, loff_t off, size_t count)
convert bitmap to list or hex format ASCII string

Parameters

bool list
indicates whether the bitmap must be list true: print in decimal list format false: print in hexadecimal bitmask format

char *buf
buffer into which string is placed

const unsigned long *maskp
pointer to bitmap to convert

int nmaskbits
size of bitmap, in bits

loff_t off
in the string from which we are copying, We copy to **buf**

size_t count
the maximum number of bytes to print

int **bitmap_pos_to_ord**(const unsigned long *buf, unsigned int pos, unsigned int nbits)
find ordinal of set bit at given position in bitmap

Parameters

const unsigned long *buf
pointer to a bitmap

unsigned int pos
a bit position in **buf** (0 <= **pos** < **nbits**)

unsigned int nbits
number of valid bit positions in **buf**

Description

Map the bit at position **pos** in **buf** (of length **nbits**) to the ordinal of which set bit it is. If it is not set or if **pos** is not a valid bit position, map to -1.

If for example, just bits 4 through 7 are set in **buf**, then **pos** values 4 through 7 will get mapped to 0 through 3, respectively, and other **pos** values will get mapped to -1. When **pos** value 7 gets mapped to (returns) **ord** value 3 in this example, that means that bit 7 is the 3rd (starting with 0th) set bit in **buf**.

The bit positions 0 through **bits** are valid positions in **buf**.

void **bitmap_onto**(unsigned long *dst, const unsigned long *orig, const unsigned long *relmap, unsigned int bits)

translate one bitmap relative to another

Parameters

unsigned long *dst
resulting translated bitmap

const unsigned long *orig
original untranslated bitmap

const unsigned long *relmap
bitmap relative to which translated

unsigned int bits
number of bits in each of these bitmaps

Description

Set the n-th bit of **dst** iff there exists some m such that the n-th bit of **relmap** is set, the m-th bit of **orig** is set, and the n-th bit of **relmap** is also the m-th _set_ bit of **relmap**. (If you understood the previous sentence the first time your read it, you're overqualified for your current job.)

In other words, **orig** is mapped onto (surjectively) **dst**, using the map { <n, m> | the n-th bit of **relmap** is the m-th set bit of **relmap** }.

Any set bits in **orig** above bit number W, where W is the weight of (number of set bits in) **relmap** are mapped nowhere. In particular, if for all bits m set in **orig**, m >= W, then **dst** will end up empty. In situations where the possibility of such an empty result is not desired, one way to avoid it is to use the *bitmap_fold()* operator, below, to first fold the **orig** bitmap over itself so that all its set bits x are in the range 0 <= x < W. The *bitmap_fold()* operator does this by setting the bit (m % W) in **dst**, for each bit (m) set in **orig**.

Example [1] for bitmap_onto():

Let's say **relmap** has bits 30-39 set, and **orig** has bits 1, 3, 5, 7, 9 and 11 set. Then on return from this routine, **dst** will have bits 31, 33, 35, 37 and 39 set.

When bit 0 is set in **orig**, it means turn on the bit in **dst** corresponding to whatever is the first bit (if any) that is turned on in **relmap**. Since bit 0 was off in the above example, we leave off that bit (bit 30) in **dst**.

When bit 1 is set in **orig** (as in the above example), it means turn on the bit in **dst** corresponding to whatever is the second bit that is turned on in **relmap**. The second bit in **relmap** that was turned on in the above example was bit 31, so we turned on bit 31 in **dst**.

Similarly, we turned on bits 33, 35, 37 and 39 in **dst**, because they were the 4th, 6th, 8th and 10th set bits set in **relmap**, and the 4th, 6th, 8th and 10th bits of **orig** (i.e. bits 3, 5, 7 and 9) were also set.

When bit 11 is set in **orig**, it means turn on the bit in **dst** corresponding to whatever is the twelfth bit that is turned on in **relmap**. In the above example, there were only ten bits turned on in **relmap** (30..39), so that bit 11 was set in **orig** had no affect on **dst**.

Example [2] for bitmap_fold() + bitmap_onto():
Let's say **relmap** has these ten bits set:

```
40 41 42 43 45 48 53 61 74 95
```

(for the curious, that's 40 plus the first ten terms of the Fibonacci sequence.)

Further lets say we use the following code, invoking *bitmap_fold()* then bitmap_onto, as suggested above to avoid the possibility of an empty **dst** result:

```
unsigned long *tmp;     // a temporary bitmap's bits

bitmap_fold(tmp, orig, bitmap_weight(relmap, bits), bits);
bitmap_onto(dst, tmp, relmap, bits);
```

Then this table shows what various values of **dst** would be, for various **orig**'s. I list the zero-based positions of each set bit. The tmp column shows the intermediate result, as computed by using *bitmap_fold()* to fold the **orig** bitmap modulo ten (the weight of **relmap**):

orig	tmp	**dst**
0	0	40
1	1	41
9	9	95
10	0	40[1]
1 3 5 7	1 3 5 7	41 43 48 61
0 1 2 3 4	0 1 2 3 4	40 41 42 43 45
0 9 18 27	0 9 8 7	40 61 74 95
0 10 20 30	0	40
0 11 22 33	0 1 2 3	40 41 42 43
0 12 24 36	0 2 4 6	40 42 45 53
78 102 211	1 2 8	41 42 74[1]

If either of **orig** or **relmap** is empty (no set bits), then **dst** will be returned empty.

If (as explained above) the only set bits in **orig** are in positions m where m >= W, (where W is the weight of **relmap**) then **dst** will once again be returned empty.

All bits in **dst** not set by the above rule are cleared.

void **bitmap_fold**(unsigned long *dst, const unsigned long *orig, unsigned int sz, unsigned int nbits)

fold larger bitmap into smaller, modulo specified size

Parameters

unsigned long *dst
resulting smaller bitmap

[1] For these marked lines, if we hadn't first done *bitmap_fold()* into tmp, then the **dst** result would have been empty.

const unsigned long *orig
original larger bitmap

unsigned int sz
specified size

unsigned int nbits
number of bits in each of these bitmaps

Description

For each bit oldbit in **orig**, set bit oldbit mod **sz** in **dst**. Clear all other bits in **dst**. See further the comment and Example [2] for *bitmap_onto()* for why and how to use this.

unsigned long **bitmap_find_next_zero_area**(unsigned long *map, unsigned long size, unsigned long start, unsigned int nr, unsigned long align_mask)

find a contiguous aligned zero area

Parameters

unsigned long *map
The address to base the search on

unsigned long size
The bitmap size in bits

unsigned long start
The bitnumber to start searching at

unsigned int nr
The number of zeroed bits we're looking for

unsigned long align_mask
Alignment mask for zero area

Description

The **align_mask** should be one less than a power of 2; the effect is that the bit offset of all zero areas this function finds is multiples of that power of 2. A **align_mask** of 0 means no alignment is required.

bool **bitmap_or_equal**(const unsigned long *src1, const unsigned long *src2, const unsigned long *src3, unsigned int nbits)

Check whether the or of two bitmaps is equal to a third

Parameters

const unsigned long *src1
Pointer to bitmap 1

const unsigned long *src2
Pointer to bitmap 2 will be or'ed with bitmap 1

const unsigned long *src3
Pointer to bitmap 3. Compare to the result of ***src1** | ***src2**

unsigned int nbits
number of bits in each of these bitmaps

Return

True if (***src1** | ***src2**) == ***src3**, false otherwise

BITMAP_FROM_U64

`BITMAP_FROM_U64 (n)`

Represent u64 value in the format suitable for bitmap.

Parameters

n
: u64 value

Description

Linux bitmaps are internally arrays of unsigned longs, i.e. 32-bit integers in 32-bit environment, and 64-bit integers in 64-bit one.

There are four combinations of endianness and length of the word in linux ABIs: LE64, BE64, LE32 and BE32.

On 64-bit kernels 64-bit LE and BE numbers are naturally ordered in bitmaps and therefore don't require any special handling.

On 32-bit kernels 32-bit LE ABI orders lo word of 64-bit number in memory prior to hi, and 32-bit BE orders hi word prior to lo. The bitmap on the other hand is represented as an array of 32-bit words and the position of bit N may therefore be calculated as: word #(N/32) and bit #(N``32``) in that word. For example, bit #42 is located at 10th position of 2nd word. It matches 32-bit LE ABI, and we can simply let the compiler store 64-bit values in memory as it usually does. But for BE we need to swap hi and lo words manually.

With all that, the macro *BITMAP_FROM_U64()* does explicit reordering of hi and lo parts of u64. For LE32 it does nothing, and for BE environment it swaps hi and lo words, as is expected by bitmap.

void **bitmap_from_u64**(unsigned long *dst, u64 mask)

Check and swap words within u64.

Parameters

unsigned long *dst
: destination bitmap

u64 mask
: source bitmap

Description

In 32-bit Big Endian kernel, when using `(u32 *)(:c:type:`val`)[*]` to read u64 mask, we will get the wrong word. That is `(u32 *)(:c:type:`val`)[0]` gets the upper 32 bits, but we expect the lower 32-bits of u64.

unsigned long **bitmap_get_value8**(const unsigned long *map, unsigned long start)

get an 8-bit value within a memory region

Parameters

const unsigned long *map
: address to the bitmap memory region

unsigned long start
bit offset of the 8-bit value; must be a multiple of 8

Description

Returns the 8-bit value located at the **start** bit offset within the **src** memory region.

void **bitmap_set_value8**(unsigned long *map, unsigned long value, unsigned long start)
set an 8-bit value within a memory region

Parameters

unsigned long *map
address to the bitmap memory region

unsigned long value
the 8-bit value; values wider than 8 bits may clobber bitmap

unsigned long start
bit offset of the 8-bit value; must be a multiple of 8

Command-line Parsing

int **get_option**(char **str, int *pint)
Parse integer from an option string

Parameters

char **str
option string

int *pint
(optional output) integer value parsed from **str**

Read an int from an option string; if available accept a subsequent comma as well.

When **pint** is NULL the function can be used as a validator of the current option in the string.

Return values: 0 - no int in string 1 - int found, no subsequent comma 2 - int found including a subsequent comma 3 - hyphen found to denote a range

Leading hyphen without integer is no integer case, but we consume it for the sake of simplification.

char ***get_options**(const char *str, int nints, int *ints)
Parse a string into a list of integers

Parameters

const char *str
String to be parsed

int nints
size of integer array

int *ints
integer array (must have room for at least one element)

This function parses a string containing a comma-separated list of integers, a hyphen-separated range of _positive_ integers, or a combination of both. The parse halts when the array is full, or when no more numbers can be retrieved from the string.

When **nints** is 0, the function just validates the given **str** and returns the amount of parseable integers as described below.

Return

The first element is filled by the number of collected integers in the range. The rest is what was parsed from the **str**.

Return value is the character in the string which caused the parse to end (typically a null terminator, if **str** is completely parseable).

unsigned long long **memparse**(const char *ptr, char **retptr)

parse a string with mem suffixes into a number

Parameters

const char *ptr
Where parse begins

char **retptr
(output) Optional pointer to next char after parse completes

Parses a string into a number. The number stored at **ptr** is potentially suffixed with K, M, G, T, P, E.

Error Pointers

IS_ERR_VALUE

IS_ERR_VALUE (x)

Detect an error pointer.

Parameters

x
The pointer to check.

Description

Like IS_ERR(), but does not generate a compiler warning if result is unused.

void ***ERR_PTR**(long error)

Create an error pointer.

Parameters

long error
A negative error code.

Description

Encodes **error** into a pointer value. Users should consider the result opaque and not assume anything about how the error is encoded.

Return

A pointer with **error** encoded within its value.

long **PTR_ERR**(__force const void *ptr)
  Extract the error code from an error pointer.

Parameters

__force const void *ptr
  An error pointer.

Return

The error code within **ptr**.

bool **IS_ERR**(__force const void *ptr)
  Detect an error pointer.

Parameters

__force const void *ptr
  The pointer to check.

Return

true if **ptr** is an error pointer, false otherwise.

bool **IS_ERR_OR_NULL**(__force const void *ptr)
  Detect an error pointer or a null pointer.

Parameters

__force const void *ptr
  The pointer to check.

Description

Like IS_ERR(), but also returns true for a null pointer.

void ***ERR_CAST**(__force const void *ptr)
  Explicitly cast an error-valued pointer to another pointer type

Parameters

__force const void *ptr
  The pointer to cast.

Description

Explicitly cast an error-valued pointer to another pointer type in such a way as to make it clear that's what's going on.

int **PTR_ERR_OR_ZERO**(__force const void *ptr)
  Extract the error code from a pointer if it has one.

Parameters

__force const void *ptr
  A potential error pointer.

Description

Convenience function that can be used inside a function that returns an error code to propagate errors received as error pointers. For example, `return PTR_ERR_OR_ZERO(ptr);` replaces:

```
if (IS_ERR(ptr))
        return PTR_ERR(ptr);
else
        return 0;
```

Return

The error code within **ptr** if it is an error pointer; 0 otherwise.

Sorting

void **sort_r**(void *base, size_t num, size_t size, cmp_r_func_t cmp_func, swap_r_func_t swap_func, const void *priv)

sort an array of elements

Parameters

void *base
pointer to data to sort

size_t num
number of elements

size_t size
size of each element

cmp_r_func_t cmp_func
pointer to comparison function

swap_r_func_t swap_func
pointer to swap function or NULL

const void *priv
third argument passed to comparison function

Description

This function does a heapsort on the given array. You may provide a swap_func function if you need to do something more than a memory copy (e.g. fix up pointers or auxiliary data), but the built-in swap avoids a slow retpoline and so is significantly faster.

Sorting time is O(n log n) both on average and worst-case. While quicksort is slightly faster on average, it suffers from exploitable O(n*n) worst-case behavior and extra memory requirements that make it less suitable for kernel use.

void **list_sort**(void *priv, struct list_head *head, list_cmp_func_t cmp)

sort a list

Parameters

void *priv
private data, opaque to *list_sort()*, passed to **cmp**

struct list_head *head
the list to sort

list_cmp_func_t cmp
the elements comparison function

Description

The comparison function **cmp** must return > 0 if **a** should sort after **b** ("**a** > **b**" if you want an ascending sort), and <= 0 if **a** should sort before **b** *or* their original order should be preserved. It is always called with the element that came first in the input in **a**, and list_sort is a stable sort, so it is not necessary to distinguish the **a** < **b** and **a** == **b** cases.

This is compatible with two styles of **cmp** function: - The traditional style which returns <0 / =0 / >0, or - Returning a boolean 0/1. The latter offers a chance to save a few cycles in the comparison (which is used by e.g. plug_ctx_cmp() in block/blk-mq.c).

A good way to write a multi-word comparison is:

```
if (a->high != b->high)
        return a->high > b->high;
if (a->middle != b->middle)
        return a->middle > b->middle;
return a->low > b->low;
```

This mergesort is as eager as possible while always performing at least 2:1 balanced merges. Given two pending sublists of size 2^k, they are merged to a size-2^(k+1) list as soon as we have 2^k following elements.

Thus, it will avoid cache thrashing as long as 3*2^k elements can fit into the cache. Not quite as good as a fully-eager bottom-up mergesort, but it does use 0.2*n fewer comparisons, so is faster in the common case that everything fits into L1.

The merging is controlled by "count", the number of elements in the pending lists. This is beautifully simple code, but rather subtle.

Each time we increment "count", we set one bit (bit k) and clear bits k-1 .. 0. Each time this happens (except the very first time for each bit, when count increments to 2^k), we merge two lists of size 2^k into one list of size 2^(k+1).

This merge happens exactly when the count reaches an odd multiple of 2^k, which is when we have 2^k elements pending in smaller lists, so it's safe to merge away two lists of size 2^k.

After this happens twice, we have created two lists of size 2^(k+1), which will be merged into a list of size 2^(k+2) before we create a third list of size 2^(k+1), so there are never more than two pending.

The number of pending lists of size 2^k is determined by the state of bit k of "count" plus two extra pieces of information:

- The state of bit k-1 (when k == 0, consider bit -1 always set), and
- Whether the higher-order bits are zero or non-zero (i.e. is count >= 2^(k+1)).

There are six states we distinguish. "x" represents some arbitrary bits, and "y" represents some arbitrary non-zero bits: 0: 00x: 0 pending of size 2^k; x pending of sizes < 2^k 1: 01x: 0 pending of size 2^k; 2^(k-1) + x pending of sizes < 2^k 2: x10x: 0 pending of size 2^k; 2^k + x pending of sizes < 2^k 3: x11x: 1 pending of size 2^k; 2^(k-1) + x pending of sizes < 2^k 4: y00x: 1 pending of size 2^k; 2^k + x pending of sizes < 2^k 5: y01x: 2 pending of size 2^k; 2^(k-1) + x pending of sizes < 2^k (merge and loop back to state 2)

We gain lists of size 2^k in the 2->3 and 4->5 transitions (because bit k-1 is set while the more significant bits are non-zero) and merge them away in the 5->2 transition. Note in particular that just before the 5->2 transition, all lower-order bits are 11 (state 3), so there is one list of each smaller size.

When we reach the end of the input, we merge all the pending lists, from smallest to largest. If you work through cases 2 to 5 above, you can see that the number of elements we merge with a list of size 2^k varies from 2^(k-1) (cases 3 and 5 when x == 0) to 2^(k+1) - 1 (second merge of case 5 when x == 2^(k-1) - 1).

Text Searching

INTRODUCTION

The textsearch infrastructure provides text searching facilities for both linear and non-linear data. Individual search algorithms are implemented in modules and chosen by the user.

ARCHITECTURE

```
  User
  +----------------+
  |        finish()|<--------------(6)-----------------+
  |get_next_block()|<--------------(5)---------------+ |
  |                |                     Algorithm   | |
  |                |                    +------------------------------+
  |                |                    |  init()   find()   destroy() |
  |                |                    +------------------------------+
  |                |       Core API           ^       ^          ^
  |                |      +---------------+  (2)     (4)        (8)
  |             (1)|----->| prepare()     |---+       |          |
  |             (3)|----->| find()/next() |-----------+          |
  |             (7)|----->| destroy()     |----------------------+
  +----------------+      +---------------+

(1) User configures a search by calling textsearch_prepare() specifying
    the search parameters such as the pattern and algorithm name.
(2) Core requests the algorithm to allocate and initialize a search
    configuration according to the specified parameters.
(3) User starts the search(es) by calling textsearch_find() or
    textsearch_next() to fetch subsequent occurrences. A state variable
    is provided to the algorithm to store persistent variables.
(4) Core eventually resets the search offset and forwards the find()
    request to the algorithm.
(5) Algorithm calls get_next_block() provided by the user continuously
    to fetch the data to be searched in block by block.
(6) Algorithm invokes finish() after the last call to get_next_block
    to clean up any leftovers from get_next_block. (Optional)
(7) User destroys the configuration by calling textsearch_destroy().
(8) Core notifies the algorithm to destroy algorithm specific
    allocations. (Optional)
```

USAGE

Before a search can be performed, a configuration must be created by calling *textsearch_prepare()* specifying the searching algorithm, the pattern to look for and flags. As a flag, you can set TS_IGNORECASE to perform case insensitive matching. But it might slow down performance of algorithm, so you should use it at own your risk. The returned configuration may then be used for an arbitrary amount of times and even in parallel as long as a separate struct ts_state variable is provided to every instance.

The actual search is performed by either calling *textsearch_find_continuous()* for linear data or by providing an own get_next_block() implementation and calling *textsearch_find()*. Both functions return the position of the first occurrence of the pattern or UINT_MAX if no match was found. Subsequent occurrences can be found by calling *textsearch_next()* regardless of the linearity of the data.

Once you're done using a configuration it must be given back via textsearch_destroy.

EXAMPLE:

```
int pos;
struct ts_config *conf;
struct ts_state state;
const char *pattern = "chicken";
const char *example = "We dance the funky chicken";

conf = textsearch_prepare("kmp", pattern, strlen(pattern),
                          GFP_KERNEL, TS_AUTOLOAD);
if (IS_ERR(conf)) {
    err = PTR_ERR(conf);
    goto errout;
}

pos = textsearch_find_continuous(conf, &state, example, strlen(example));
if (pos != UINT_MAX)
    panic("Oh my god, dancing chickens at %d\n", pos);

textsearch_destroy(conf);
```

int **textsearch_register**(struct ts_ops *ops)

register a textsearch module

Parameters

struct ts_ops *ops
operations lookup table

Description

This function must be called by textsearch modules to announce their presence. The specified &**ops** must have name set to a unique identifier and the callbacks find(), init(), get_pattern(), and get_pattern_len() must be implemented.

Returns 0 or -EEXISTS if another module has already registered with same name.

int **textsearch_unregister**(struct ts_ops *ops)

unregister a textsearch module

Parameters

struct ts_ops *ops
: operations lookup table

Description

This function must be called by textsearch modules to announce their disappearance for examples when the module gets unloaded. The `ops` parameter must be the same as the one during the registration.

Returns 0 on success or -ENOENT if no matching textsearch registration was found.

unsigned int **textsearch_find_continuous**(struct ts_config *conf, struct ts_state *state, const void *data, unsigned int len)

search a pattern in continuous/linear data

Parameters

struct ts_config *conf
: search configuration

struct ts_state *state
: search state

const void *data
: data to search in

unsigned int len
: length of data

Description

A simplified version of *textsearch_find()* for continuous/linear data. Call *textsearch_next()* to retrieve subsequent matches.

Returns the position of first occurrence of the pattern or `UINT_MAX` if no occurrence was found.

struct ts_config ***textsearch_prepare**(const char *algo, const void *pattern, unsigned int len, gfp_t gfp_mask, int flags)

Prepare a search

Parameters

const char *algo
: name of search algorithm

const void *pattern
: pattern data

unsigned int len
: length of pattern

gfp_t gfp_mask
: allocation mask

int flags
: search flags

Description

Looks up the search algorithm module and creates a new textsearch configuration for the specified pattern.

Returns a new textsearch configuration according to the specified parameters or a ERR_PTR(). If a zero length pattern is passed, this function returns EINVAL.

Note

The format of the pattern may not be compatible between
the various search algorithms.

void **textsearch_destroy**(struct ts_config *conf)
destroy a search configuration

Parameters

struct ts_config *conf
search configuration

Description

Releases all references of the configuration and frees up the memory.

unsigned int **textsearch_next**(struct ts_config *conf, struct ts_state *state)
continue searching for a pattern

Parameters

struct ts_config *conf
search configuration

struct ts_state *state
search state

Description

Continues a search looking for more occurrences of the pattern. *textsearch_find()* must be called to find the first occurrence in order to reset the state.

Returns the position of the next occurrence of the pattern or UINT_MAX if not match was found.

unsigned int **textsearch_find**(struct ts_config *conf, struct ts_state *state)
start searching for a pattern

Parameters

struct ts_config *conf
search configuration

struct ts_state *state
search state

Description

Returns the position of first occurrence of the pattern or UINT_MAX if no match was found.

void ***textsearch_get_pattern**(struct ts_config *conf)
return head of the pattern

Parameters

struct ts_config *conf
search configuration

unsigned int **textsearch_get_pattern_len**(struct ts_config *conf)
return length of the pattern

Parameters

struct ts_config *conf
search configuration

1.1.4 CRC and Math Functions in Linux

Arithmetic Overflow Checking

check_add_overflow

`check_add_overflow (a, b, d)`

Calculate addition with overflow checking

Parameters

a
first addend

b
second addend

d
pointer to store sum

Description

Returns 0 on success.

***d** holds the results of the attempted addition, but is not considered "safe for use" on a non-zero return value, which indicates that the sum has overflowed or been truncated.

check_sub_overflow

`check_sub_overflow (a, b, d)`

Calculate subtraction with overflow checking

Parameters

a
minuend; value to subtract from

b
subtrahend; value to subtract from **a**

d
pointer to store difference

Description

Returns 0 on success.

***d** holds the results of the attempted subtraction, but is not considered "safe for use" on a non-zero return value, which indicates that the difference has underflowed or been truncated.

check_mul_overflow

`check_mul_overflow (a, b, d)`

Calculate multiplication with overflow checking

Parameters

a
first factor

b
second factor

d
pointer to store product

Description

Returns 0 on success.

***d** holds the results of the attempted multiplication, but is not considered "safe for use" on a non-zero return value, which indicates that the product has overflowed or been truncated.

check_shl_overflow

`check_shl_overflow (a, s, d)`

Calculate a left-shifted value and check overflow

Parameters

a
Value to be shifted

s
How many bits left to shift

d
Pointer to where to store the result

Description

Computes ***d** = (**a** << **s**)

Returns true if '***d**' cannot hold the result or when '**a** << **s**' doesn't make sense. Example conditions:

- '**a** << **s**' causes bits to be lost when stored in ***d**.
- '**s**' is garbage (e.g. negative) or so large that the result of '**a** << **s**' is guaranteed to be 0.
- '**a**' is negative.
- '**a** << **s**' sets the sign bit, if any, in '***d**'.

'***d**' will hold the results of the attempted shift, but is not considered "safe for use" if true is returned.

overflows_type

`overflows_type (n, T)`

helper for checking the overflows between value, variables, or data type

Parameters

n
: source constant value or variable to be checked

T
: destination variable or data type proposed to store **x**

Description

Compares the **x** expression for whether or not it can safely fit in the storage of the type in **T**. **x** and **T** can have different types. If **x** is a constant expression, this will also resolve to a constant expression.

Return

true if overflow can occur, false otherwise.

castable_to_type

`castable_to_type (n, T)`

like __same_type(), but also allows for casted literals

Parameters

n
: variable or constant value

T
: variable or data type

Description

Unlike the __same_type() macro, this allows a constant value as the first argument. If this value would not overflow into an assignment of the second argument's type, it returns true. Otherwise, this falls back to __same_type().

size_t **size_mul**(size_t factor1, size_t factor2)

Calculate size_t multiplication with saturation at SIZE_MAX

Parameters

size_t factor1
: first factor

size_t factor2
: second factor

Return

calculate **factor1** * **factor2**, both promoted to size_t, with any overflow causing the return value to be SIZE_MAX. The lvalue must be size_t to avoid implicit type conversion.

size_t **size_add**(size_t addend1, size_t addend2)

Calculate size_t addition with saturation at SIZE_MAX

Parameters

size_t addend1
: first addend

size_t addend2
  second addend

Return

calculate **addend1 + addend2**, both promoted to size_t, with any overflow causing the return value to be SIZE_MAX. The lvalue must be size_t to avoid implicit type conversion.

size_t **size_sub**(size_t minuend, size_t subtrahend)
  Calculate size_t subtraction with saturation at SIZE_MAX

Parameters

size_t minuend
  value to subtract from

size_t subtrahend
  value to subtract from **minuend**

Return

calculate **minuend - subtrahend**, both promoted to size_t, with any overflow causing the return value to be SIZE_MAX. For composition with the size_add() and size_mul() helpers, neither argument may be SIZE_MAX (or the result with be forced to SIZE_MAX). The lvalue must be size_t to avoid implicit type conversion.

array_size

`array_size (a, b)`

  Calculate size of 2-dimensional array.

Parameters

a
  dimension one

b
  dimension two

Description

Calculates size of 2-dimensional array: **a** * **b**.

Return

number of bytes needed to represent the array or SIZE_MAX on overflow.

array3_size

`array3_size (a, b, c)`

  Calculate size of 3-dimensional array.

Parameters

a
  dimension one

b
  dimension two

c
    dimension three

Description

Calculates size of 3-dimensional array: **a** * **b** * **c**.

Return

number of bytes needed to represent the array or SIZE_MAX on overflow.

flex_array_size

`flex_array_size (p, member, count)`

Calculate size of a flexible array member within an enclosing structure.

Parameters

p
    Pointer to the structure.

member
    Name of the flexible array member.

count
    Number of elements in the array.

Description

Calculates size of a flexible array of **count** number of **member** elements, at the end of structure **p**.

Return

number of bytes needed or SIZE_MAX on overflow.

struct_size

`struct_size (p, member, count)`

Calculate size of structure with trailing flexible array.

Parameters

p
    Pointer to the structure.

member
    Name of the array member.

count
    Number of elements in the array.

Description

Calculates size of memory needed for structure of **p** followed by an array of **count** number of **member** elements.

Return

number of bytes needed or SIZE_MAX on overflow.

struct_size_t

`struct_size_t (type, member, count)`

Calculate size of structure with trailing flexible array

Parameters

type
structure type name.

member
Name of the array member.

count
Number of elements in the array.

Description

Calculates size of memory needed for structure **type** followed by an array of **count** number of **member** elements. Prefer using struct_size() when possible instead, to keep calculations associated with a specific instance variable of type **type**.

Return

number of bytes needed or SIZE_MAX on overflow.

CRC Functions

uint8_t **crc4**(uint8_t c, uint64_t x, int bits)
calculate the 4-bit crc of a value.

Parameters

uint8_t c
starting crc4

uint64_t x
value to checksum

int bits
number of bits in **x** to checksum

Description

Returns the crc4 value of **x**, using polynomial 0b10111.

The **x** value is treated as left-aligned, and bits above **bits** are ignored in the crc calculations.

u8 **crc7_be**(u8 crc, const u8 *buffer, size_t len)
update the CRC7 for the data buffer

Parameters

u8 crc
previous CRC7 value

const u8 *buffer
data pointer

size_t len
number of bytes in the buffer

Context

any

Description

Returns the updated CRC7 value. The CRC7 is left-aligned in the byte (the lsbit is always 0), as that makes the computation easier, and all callers want it in that form.

void **crc8_populate_msb**(u8 table[CRC8_TABLE_SIZE], u8 polynomial)
fill crc table for given polynomial in reverse bit order.

Parameters

u8 table[CRC8_TABLE_SIZE]
table to be filled.

u8 polynomial
polynomial for which table is to be filled.

void **crc8_populate_lsb**(u8 table[CRC8_TABLE_SIZE], u8 polynomial)
fill crc table for given polynomial in regular bit order.

Parameters

u8 table[CRC8_TABLE_SIZE]
table to be filled.

u8 polynomial
polynomial for which table is to be filled.

u8 **crc8**(const u8 table[CRC8_TABLE_SIZE], const u8 *pdata, size_t nbytes, u8 crc)
calculate a crc8 over the given input data.

Parameters

const u8 table[CRC8_TABLE_SIZE]
crc table used for calculation.

const u8 *pdata
pointer to data buffer.

size_t nbytes
number of bytes in data buffer.

u8 crc
previous returned crc8 value.

u16 **crc16**(u16 crc, u8 const *buffer, size_t len)
compute the CRC-16 for the data buffer

Parameters

u16 crc
previous CRC value

u8 const *buffer
data pointer

size_t len
number of bytes in the buffer

Description

Returns the updated CRC value.

u32 __pure **crc32_le_generic**(u32 crc, unsigned char const *p, size_t len, const u32 (*tab)[256], u32 polynomial)

Calculate bitwise little-endian Ethernet AUTODIN II CRC32/CRC32C

Parameters

u32 crc
seed value for computation. ~0 for Ethernet, sometimes 0 for other uses, or the previous crc32/crc32c value if computing incrementally.

unsigned char const *p
pointer to buffer over which CRC32/CRC32C is run

size_t len
length of buffer **p**

const u32 (*tab)[256]
little-endian Ethernet table

u32 polynomial
CRC32/CRC32c LE polynomial

u32 **crc32_generic_shift**(u32 crc, size_t len, u32 polynomial)

Append **len** 0 bytes to crc, in logarithmic time

Parameters

u32 crc
The original little-endian CRC (i.e. lsbit is x^31 coefficient)

size_t len
The number of bytes. **crc** is multiplied by x^(8***len**)

u32 polynomial
The modulus used to reduce the result to 32 bits.

Description

It's possible to parallelize CRC computations by computing a CRC over separate ranges of a buffer, then summing them. This shifts the given CRC by 8*len bits (i.e. produces the same effect as appending len bytes of zero to the data), in time proportional to log(len).

u32 __pure **crc32_be_generic**(u32 crc, unsigned char const *p, size_t len, const u32 (*tab)[256], u32 polynomial)

Calculate bitwise big-endian Ethernet AUTODIN II CRC32

Parameters

u32 crc
seed value for computation. ~0 for Ethernet, sometimes 0 for other uses, or the previous crc32 value if computing incrementally.

unsigned char const *p
pointer to buffer over which CRC32 is run

size_t len
length of buffer **p**

const u32 (*tab)[256]
big-endian Ethernet table

u32 polynomial
CRC32 BE polynomial

u16 **crc_ccitt**(u16 crc, u8 const *buffer, size_t len)
recompute the CRC (CRC-CCITT variant) for the data buffer

Parameters

u16 crc
previous CRC value

u8 const *buffer
data pointer

size_t len
number of bytes in the buffer

u16 **crc_ccitt_false**(u16 crc, u8 const *buffer, size_t len)
recompute the CRC (CRC-CCITT-FALSE variant) for the data buffer

Parameters

u16 crc
previous CRC value

u8 const *buffer
data pointer

size_t len
number of bytes in the buffer

u16 **crc_itu_t**(u16 crc, const u8 *buffer, size_t len)
Compute the CRC-ITU-T for the data buffer

Parameters

u16 crc
previous CRC value

const u8 *buffer
data pointer

size_t len
number of bytes in the buffer

Description

Returns the updated CRC value

Base 2 log and power Functions

bool **is_power_of_2**(unsigned long n)

check if a value is a power of two

Parameters

unsigned long n
the value to check

Description

Determine whether some value is a power of two, where zero is *not* considered a power of two.

Return

true if **n** is a power of 2, otherwise false.

unsigned long **__roundup_pow_of_two**(unsigned long n)

round up to nearest power of two

Parameters

unsigned long n
value to round up

unsigned long **__rounddown_pow_of_two**(unsigned long n)

round down to nearest power of two

Parameters

unsigned long n
value to round down

const_ilog2

`const_ilog2 (n)`

log base 2 of 32-bit or a 64-bit constant unsigned value

Parameters

n
parameter

Description

Use this where sparse expects a true constant expression, e.g. for array indices.

ilog2

`ilog2 (n)`

log base 2 of 32-bit or a 64-bit unsigned value

Parameters

n
parameter

Description

constant-capable log of base 2 calculation - this can be used to initialise global variables from constant data, hence the massive ternary operator construction

selects the appropriately-sized optimised version depending on sizeof(n)

roundup_pow_of_two

`roundup_pow_of_two (n)`

round the given value up to nearest power of two

Parameters

n
parameter

Description

round the given value up to the nearest power of two - the result is undefined when n == 0 - this can be used to initialise global variables from constant data

rounddown_pow_of_two

`rounddown_pow_of_two (n)`

round the given value down to nearest power of two

Parameters

n
parameter

Description

round the given value down to the nearest power of two - the result is undefined when n == 0 - this can be used to initialise global variables from constant data

order_base_2

`order_base_2 (n)`

calculate the (rounded up) base 2 order of the argument

Parameters

n
parameter

Description

The first few values calculated by this routine:
ob2(0) = 0 ob2(1) = 0 ob2(2) = 1 ob2(3) = 2 ob2(4) = 2 ob2(5) = 3 … and so on.

bits_per

`bits_per (n)`

calculate the number of bits required for the argument

Parameters

n
parameter

Description

This is constant-capable and can be used for compile time initializations, e.g bitfields.

The first few values calculated by this routine: bf(0) = 1 bf(1) = 1 bf(2) = 2 bf(3) = 2 bf(4) = 3 … and so on.

Integer log and power Functions

unsigned int **intlog2**(u32 value)
computes log2 of a value; the result is shifted left by 24 bits

Parameters

u32 value
The value (must be != 0)

Description

to use rational values you can use the following method:

intlog2(value) = intlog2(value * 2^x) - x * 2^24

Some usecase examples:

intlog2(8) will give 3 << 24 = 3 * 2^24

intlog2(9) will give 3 << 24 + … = 3.16… * 2^24

intlog2(1.5) = intlog2(3) - 2^24 = 0.584… * 2^24

Return

log2(value) * 2^24

unsigned int **intlog10**(u32 value)
computes log10 of a value; the result is shifted left by 24 bits

Parameters

u32 value
The value (must be != 0)

Description

to use rational values you can use the following method:

intlog10(value) = intlog10(value * 10^x) - x * 2^24

An usecase example:

intlog10(1000) will give 3 << 24 = 3 * 2^24

due to the implementation intlog10(1000) might be not exactly 3 * 2^24

look at intlog2 for similar examples

Return

log10(value) * 2^24

u64 **int_pow**(u64 base, unsigned int exp)
    computes the exponentiation of the given base and exponent

Parameters

u64 base
    base which will be raised to the given power

unsigned int exp
    power to be raised to

Description

Computes: pow(base, exp), i.e. **base** raised to the **exp** power

unsigned long **int_sqrt**(unsigned long x)
    computes the integer square root

Parameters

unsigned long x
    integer of which to calculate the sqrt

Description

Computes: floor(sqrt(x))

u32 **int_sqrt64**(u64 x)
    strongly typed int_sqrt function when minimum 64 bit input is expected.

Parameters

u64 x
    64bit integer of which to calculate the sqrt

Division Functions

do_div

`do_div (n, base)`

    returns 2 values: calculate remainder and update new dividend

Parameters

n
    uint64_t dividend (will be updated)

base
    uint32_t divisor

Description

Summary: `uint32_t remainder = n % base; n = n / base;`

Return

(uint32_t)remainder

NOTE

macro parameter **n** is evaluated multiple times, beware of side effects!

u64 **div_u64_rem**(u64 dividend, u32 divisor, u32 *remainder)
unsigned 64bit divide with 32bit divisor with remainder

Parameters

u64 dividend
unsigned 64bit dividend

u32 divisor
unsigned 32bit divisor

u32 *remainder
pointer to unsigned 32bit remainder

Return

sets `*remainder`, then returns dividend / divisor

Description

This is commonly provided by 32bit archs to provide an optimized 64bit divide.

s64 **div_s64_rem**(s64 dividend, s32 divisor, s32 *remainder)
signed 64bit divide with 32bit divisor with remainder

Parameters

s64 dividend
signed 64bit dividend

s32 divisor
signed 32bit divisor

s32 *remainder
pointer to signed 32bit remainder

Return

sets `*remainder`, then returns dividend / divisor

u64 **div64_u64_rem**(u64 dividend, u64 divisor, u64 *remainder)
unsigned 64bit divide with 64bit divisor and remainder

Parameters

u64 dividend
unsigned 64bit dividend

u64 divisor
unsigned 64bit divisor

u64 *remainder
pointer to unsigned 64bit remainder

Return

sets `*remainder`, then returns dividend / divisor

u64 **div64_u64**(u64 dividend, u64 divisor)
unsigned 64bit divide with 64bit divisor

Parameters

u64 dividend
unsigned 64bit dividend

u64 divisor
unsigned 64bit divisor

Return

dividend / divisor

s64 **div64_s64**(s64 dividend, s64 divisor)
signed 64bit divide with 64bit divisor

Parameters

s64 dividend
signed 64bit dividend

s64 divisor
signed 64bit divisor

Return

dividend / divisor

u64 **div_u64**(u64 dividend, u32 divisor)
unsigned 64bit divide with 32bit divisor

Parameters

u64 dividend
unsigned 64bit dividend

u32 divisor
unsigned 32bit divisor

Description

This is the most common 64bit divide and should be used if possible, as many 32bit archs can optimize this variant better than a full 64bit divide.

Return

dividend / divisor

s64 **div_s64**(s64 dividend, s32 divisor)
signed 64bit divide with 32bit divisor

Parameters

s64 dividend
signed 64bit dividend

s32 divisor
signed 32bit divisor

Return

dividend / divisor

DIV64_U64_ROUND_UP

`DIV64_U64_ROUND_UP (ll, d)`

unsigned 64bit divide with 64bit divisor rounded up

Parameters

ll
unsigned 64bit dividend

d
unsigned 64bit divisor

Description

Divide unsigned 64bit dividend by unsigned 64bit divisor and round up.

Return

dividend / divisor rounded up

DIV64_U64_ROUND_CLOSEST

`DIV64_U64_ROUND_CLOSEST (dividend, divisor)`

unsigned 64bit divide with 64bit divisor rounded to nearest integer

Parameters

dividend
unsigned 64bit dividend

divisor
unsigned 64bit divisor

Description

Divide unsigned 64bit dividend by unsigned 64bit divisor and round to closest integer.

Return

dividend / divisor rounded to nearest integer

DIV_U64_ROUND_CLOSEST

`DIV_U64_ROUND_CLOSEST (dividend, divisor)`

unsigned 64bit divide with 32bit divisor rounded to nearest integer

Parameters

dividend
unsigned 64bit dividend

divisor
unsigned 32bit divisor

Description

Divide unsigned 64bit dividend by unsigned 32bit divisor and round to closest integer.

Return

dividend / divisor rounded to nearest integer

DIV_S64_ROUND_CLOSEST

`DIV_S64_ROUND_CLOSEST (dividend, divisor)`

signed 64bit divide with 32bit divisor rounded to nearest integer

Parameters

dividend
signed 64bit dividend

divisor
signed 32bit divisor

Description

Divide signed 64bit dividend by signed 32bit divisor and round to closest integer.

Return

dividend / divisor rounded to nearest integer

unsigned long **gcd**(unsigned long a, unsigned long b)

calculate and return the greatest common divisor of 2 unsigned longs

Parameters

unsigned long a
first value

unsigned long b
second value

UUID/GUID

void **generate_random_uuid**(unsigned char uuid[16])

generate a random UUID

Parameters

unsigned char uuid[16]
where to put the generated UUID

Description

Random UUID interface

Used to create a Boot ID or a filesystem UUID/GUID, but can be useful for other kernel drivers.

bool **uuid_is_valid**(const char *uuid)

checks if a UUID string is valid

Parameters

const char *uuid
UUID string to check

Description

It checks if the UUID string is following the format:
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

where x is a hex digit.

Return

true if input is valid UUID string.

1.1.5 Kernel IPC facilities

IPC utilities

int **ipc_init**(void)

initialise ipc subsystem

Parameters

void
no arguments

Description

The various sysv ipc resources (semaphores, messages and shared memory) are initialised.

A callback routine is registered into the memory hotplug notifier chain: since msgmni scales to lowmem this callback routine will be called upon successful memory add / remove to recompute msmgni.

void **ipc_init_ids**(struct ipc_ids *ids)

initialise ipc identifiers

Parameters

struct ipc_ids *ids
ipc identifier set

Description

Set up the sequence range to use for the ipc identifier range (limited below ipc_mni) then initialise the keys hashtable and ids idr.

void **ipc_init_proc_interface**(const char *path, const char *header, int ids, int (*show)(struct seq_file*, void*))

create a proc interface for sysipc types using a seq_file interface.

Parameters

const char *path
Path in procfs

const char *header
Banner to be printed at the beginning of the file.

int ids
ipc id table to iterate.

int (*show)(struct seq_file *, void *)
show routine.

struct kern_ipc_perm ***ipc_findkey**(struct ipc_ids *ids, key_t key)

find a key in an ipc identifier set

Parameters

struct ipc_ids *ids
ipc identifier set

key_t key
key to find

Description

Returns the locked pointer to the ipc structure if found or NULL otherwise. If key is found ipc points to the owning ipc structure

Called with writer ipc_ids.rwsem held.

int **ipc_addid**(struct ipc_ids *ids, struct kern_ipc_perm *new, int limit)

add an ipc identifier

Parameters

struct ipc_ids *ids
ipc identifier set

struct kern_ipc_perm *new
new ipc permission set

int limit
limit for the number of used ids

Description

Add an entry 'new' to the ipc ids idr. The permissions object is initialised and the first free entry is set up and the index assigned is returned. The 'new' entry is returned in a locked state on success.

On failure the entry is not locked and a negative err-code is returned. The caller must use ipc_rcu_putref() to free the identifier.

Called with writer ipc_ids.rwsem held.

int **ipcget_new**(struct ipc_namespace *ns, struct ipc_ids *ids, const struct ipc_ops *ops, struct ipc_params *params)

create a new ipc object

Parameters

struct ipc_namespace *ns
ipc namespace

struct ipc_ids *ids
ipc identifier set

const struct ipc_ops *ops
the actual creation routine to call

struct ipc_params *params
its parameters

Description

This routine is called by sys_msgget, sys_semget() and sys_shmget() when the key is IPC_PRIVATE.

int **ipc_check_perms**(struct ipc_namespace *ns, struct kern_ipc_perm *ipcp, const struct ipc_ops *ops, struct ipc_params *params)

check security and permissions for an ipc object

Parameters

struct ipc_namespace *ns
ipc namespace

struct kern_ipc_perm *ipcp
ipc permission set

const struct ipc_ops *ops
the actual security routine to call

struct ipc_params *params
its parameters

Description

This routine is called by sys_msgget(), sys_semget() and sys_shmget() when the key is not IPC_PRIVATE and that key already exists in the ds IDR.

On success, the ipc id is returned.

It is called with ipc_ids.rwsem and ipcp->lock held.

int **ipcget_public**(struct ipc_namespace *ns, struct ipc_ids *ids, const struct ipc_ops *ops, struct ipc_params *params)

get an ipc object or create a new one

Parameters

struct ipc_namespace *ns
ipc namespace

struct ipc_ids *ids
ipc identifier set

const struct ipc_ops *ops
the actual creation routine to call

struct ipc_params *params
its parameters

Description

This routine is called by sys_msgget, sys_semget() and sys_shmget() when the key is not IPC_PRIVATE. It adds a new entry if the key is not found and does some permission / security checkings if the key is found.

On success, the ipc id is returned.

void **ipc_kht_remove**(struct ipc_ids *ids, struct kern_ipc_perm *ipcp)

remove an ipc from the key hashtable

Parameters

struct ipc_ids *ids
    ipc identifier set

struct kern_ipc_perm *ipcp
    ipc perm structure containing the key to remove

Description

ipc_ids.rwsem (as a writer) and the spinlock for this ID are held before this function is called, and remain locked on the exit.

int **ipc_search_maxidx**(struct ipc_ids *ids, int limit)
    search for the highest assigned index

Parameters

struct ipc_ids *ids
    ipc identifier set

int limit
    known upper limit for highest assigned index

Description

The function determines the highest assigned index in **ids**. It is intended to be called when ids->max_idx needs to be updated. Updating ids->max_idx is necessary when the current highest index ipc object is deleted. If no ipc object is allocated, then -1 is returned.

ipc_ids.rwsem needs to be held by the caller.

void **ipc_rmid**(struct ipc_ids *ids, struct kern_ipc_perm *ipcp)
    remove an ipc identifier

Parameters

struct ipc_ids *ids
    ipc identifier set

struct kern_ipc_perm *ipcp
    ipc perm structure containing the identifier to remove

Description

ipc_ids.rwsem (as a writer) and the spinlock for this ID are held before this function is called, and remain locked on the exit.

void **ipc_set_key_private**(struct ipc_ids *ids, struct kern_ipc_perm *ipcp)
    switch the key of an existing ipc to IPC_PRIVATE

Parameters

struct ipc_ids *ids
    ipc identifier set

struct kern_ipc_perm *ipcp
    ipc perm structure containing the key to modify

Description

ipc_ids.rwsem (as a writer) and the spinlock for this ID are held before this function is called, and remain locked on the exit.

int **ipcperms**(struct ipc_namespace *ns, struct kern_ipc_perm *ipcp, short flag)
check ipc permissions

Parameters

struct ipc_namespace *ns
ipc namespace

struct kern_ipc_perm *ipcp
ipc permission set

short flag
desired permission set

Description

Check user, group, other permissions for access to ipc resources. return 0 if allowed

flag will most probably be 0 or `S_...UGO` from <linux/stat.h>

void **kernel_to_ipc64_perm**(struct kern_ipc_perm *in, struct ipc64_perm *out)
convert kernel ipc permissions to user

Parameters

struct kern_ipc_perm *in
kernel permissions

struct ipc64_perm *out
new style ipc permissions

Description

Turn the kernel object **in** into a set of permissions descriptions for returning to userspace (**out**).

void **ipc64_perm_to_ipc_perm**(struct ipc64_perm *in, struct ipc_perm *out)
convert new ipc permissions to old

Parameters

struct ipc64_perm *in
new style ipc permissions

struct ipc_perm *out
old style ipc permissions

Description

Turn the new style permissions object **in** into a compatibility object and store it into the **out** pointer.

struct kern_ipc_perm ***ipc_obtain_object_idr**(struct ipc_ids *ids, int id)

Parameters

struct ipc_ids *ids
ipc identifier set

int id
ipc id to look for

Description

Look for an id in the ipc ids idr and return associated ipc object.

Call inside the RCU critical section. The ipc object is *not* locked on exit.

struct kern_ipc_perm ***ipc_obtain_object_check**(struct ipc_ids *ids, int id)

Parameters

struct ipc_ids *ids
ipc identifier set

int id
ipc id to look for

Description

Similar to *ipc_obtain_object_idr()* but also checks the ipc object sequence number.

Call inside the RCU critical section. The ipc object is *not* locked on exit.

int **ipcget**(struct ipc_namespace *ns, struct ipc_ids *ids, const struct ipc_ops *ops, struct ipc_params *params)
Common sys_*get() code

Parameters

struct ipc_namespace *ns
namespace

struct ipc_ids *ids
ipc identifier set

const struct ipc_ops *ops
operations to be called on ipc object creation, permission checks and further checks

struct ipc_params *params
the parameters needed by the previous operations.

Description

Common routine called by sys_msgget(), sys_semget() and sys_shmget().

int **ipc_update_perm**(struct ipc64_perm *in, struct kern_ipc_perm *out)
update the permissions of an ipc object

Parameters

struct ipc64_perm *in
the permission given as input.

struct kern_ipc_perm *out
the permission of the ipc to set.

struct kern_ipc_perm ***ipcctl_obtain_check**(struct ipc_namespace *ns, struct ipc_ids *ids, int id, int cmd, struct ipc64_perm *perm, int extra_perm)
retrieve an ipc object and check permissions

Parameters

struct ipc_namespace *ns
ipc namespace

struct ipc_ids *ids
the table of ids where to look for the ipc

int id
the id of the ipc to retrieve

int cmd
the cmd to check

struct ipc64_perm *perm
the permission to set

int extra_perm
one extra permission parameter used by msq

Description

This function does some common audit and permissions check for some IPC_XXX cmd and is called from semctl_down, shmctl_down and msgctl_down.

It:

- retrieves the ipc object with the given id in the given table.
- performs some audit and permission check, depending on the given cmd
- returns a pointer to the ipc object or otherwise, the corresponding error.

Call holding the both the rwsem and the rcu read lock.

int **ipc_parse_version**(int *cmd)
ipc call version

Parameters

int *cmd
pointer to command

Description

Return IPC_64 for new style IPC and IPC_OLD for old style IPC. The **cmd** value is turned from an encoding command and version into just the command code.

struct kern_ipc_perm ***sysvipc_find_ipc**(struct ipc_ids *ids, loff_t *pos)
Find and lock the ipc structure based on seq pos

Parameters

struct ipc_ids *ids
ipc identifier set

loff_t *pos
expected position

Description

The function finds an ipc structure, based on the sequence file position **pos**. If there is no ipc structure at position **pos**, then the successor is selected. If a structure is found, then it is locked (both rcu_read_lock() and ipc_lock_object()) and **pos** is set to the position needed to locate the found ipc structure. If nothing is found (i.e. EOF), **pos** is not modified.

The function returns the found ipc structure, or NULL at EOF.

1.1.6 FIFO Buffer

kfifo interface

DECLARE_KFIFO_PTR

`DECLARE_KFIFO_PTR (fifo, type)`

macro to declare a fifo pointer object

Parameters

fifo
name of the declared fifo

type
type of the fifo elements

DECLARE_KFIFO

`DECLARE_KFIFO (fifo, type, size)`

macro to declare a fifo object

Parameters

fifo
name of the declared fifo

type
type of the fifo elements

size
the number of elements in the fifo, this must be a power of 2

INIT_KFIFO

`INIT_KFIFO (fifo)`

Initialize a fifo declared by DECLARE_KFIFO

Parameters

fifo
name of the declared fifo datatype

DEFINE_KFIFO

`DEFINE_KFIFO (fifo, type, size)`

macro to define and initialize a fifo

Parameters

fifo
name of the declared fifo datatype

type
    type of the fifo elements

size
    the number of elements in the fifo, this must be a power of 2

Note

the macro can be used for global and local fifo data type variables.

kfifo_initialized

`kfifo_initialized (fifo)`

Check if the fifo is initialized

Parameters

fifo
    address of the fifo to check

Description

Return `true` if fifo is initialized, otherwise `false`. Assumes the fifo was 0 before.

kfifo_esize

`kfifo_esize (fifo)`

returns the size of the element managed by the fifo

Parameters

fifo
    address of the fifo to be used

kfifo_recsize

`kfifo_recsize (fifo)`

returns the size of the record length field

Parameters

fifo
    address of the fifo to be used

kfifo_size

`kfifo_size (fifo)`

returns the size of the fifo in elements

Parameters

fifo
    address of the fifo to be used

kfifo_reset

`kfifo_reset (fifo)`

removes the entire fifo content

Parameters

fifo
address of the fifo to be used

Note

usage of *kfifo_reset()* is dangerous. It should be only called when the fifo is exclusived locked or when it is secured that no other thread is accessing the fifo.

kfifo_reset_out

`kfifo_reset_out (fifo)`

skip fifo content

Parameters

fifo
address of the fifo to be used

Note

The usage of *kfifo_reset_out()* is safe until it will be only called from the reader thread and there is only one concurrent reader. Otherwise it is dangerous and must be handled in the same way as *kfifo_reset()*.

kfifo_len

`kfifo_len (fifo)`

returns the number of used elements in the fifo

Parameters

fifo
address of the fifo to be used

kfifo_is_empty

`kfifo_is_empty (fifo)`

returns true if the fifo is empty

Parameters

fifo
address of the fifo to be used

kfifo_is_empty_spinlocked

`kfifo_is_empty_spinlocked (fifo, lock)`

returns true if the fifo is empty using a spinlock for locking

Parameters

fifo
address of the fifo to be used

lock
spinlock to be used for locking

kfifo_is_empty_spinlocked_noirqsave

`kfifo_is_empty_spinlocked_noirqsave (fifo, lock)`

returns true if the fifo is empty using a spinlock for locking, doesn't disable interrupts

Parameters

fifo
address of the fifo to be used

lock
spinlock to be used for locking

kfifo_is_full

`kfifo_is_full (fifo)`

returns true if the fifo is full

Parameters

fifo
address of the fifo to be used

kfifo_avail

`kfifo_avail (fifo)`

returns the number of unused elements in the fifo

Parameters

fifo
address of the fifo to be used

kfifo_skip

`kfifo_skip (fifo)`

skip output data

Parameters

fifo
address of the fifo to be used

kfifo_peek_len

`kfifo_peek_len (fifo)`

gets the size of the next fifo record

Parameters

fifo
address of the fifo to be used

Description

This function returns the size of the next fifo record in number of bytes.

kfifo_alloc

`kfifo_alloc (fifo, size, gfp_mask)`

dynamically allocates a new fifo buffer

Parameters

fifo
pointer to the fifo

size
the number of elements in the fifo, this must be a power of 2

gfp_mask
get_free_pages mask, passed to kmalloc()

Description

This macro dynamically allocates a new fifo buffer.

The number of elements will be rounded-up to a power of 2. The fifo will be release with *kfifo_free()*. Return 0 if no error, otherwise an error code.

kfifo_free

`kfifo_free (fifo)`

frees the fifo

Parameters

fifo
the fifo to be freed

kfifo_init

`kfifo_init (fifo, buffer, size)`

initialize a fifo using a preallocated buffer

Parameters

fifo
the fifo to assign the buffer

buffer
the preallocated buffer to be used

size
the size of the internal buffer, this have to be a power of 2

Description

This macro initializes a fifo using a preallocated buffer.

The number of elements will be rounded-up to a power of 2. Return 0 if no error, otherwise an error code.

kfifo_put

`kfifo_put (fifo, val)`

put data into the fifo

Parameters

fifo
address of the fifo to be used

val
the data to be added

Description

This macro copies the given value into the fifo. It returns 0 if the fifo was full. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_get

`kfifo_get (fifo, val)`

get data from the fifo

Parameters

fifo
address of the fifo to be used

val
address where to store the data

Description

This macro reads the data from the fifo. It returns 0 if the fifo was empty. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_peek

`kfifo_peek (fifo, val)`

get data from the fifo without removing

Parameters

fifo
address of the fifo to be used

val
address where to store the data

Description

This reads the data from the fifo without removing it from the fifo. It returns 0 if the fifo was empty. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_in

`kfifo_in (fifo, buf, n)`

put data into the fifo

Parameters

fifo
address of the fifo to be used

buf
the data to be added

n
number of elements to be added

Description

This macro copies the given buffer into the fifo and returns the number of copied elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_in_spinlocked

`kfifo_in_spinlocked (fifo, buf, n, lock)`

put data into the fifo using a spinlock for locking

Parameters

fifo
address of the fifo to be used

buf
the data to be added

n
number of elements to be added

lock
pointer to the spinlock to use for locking

Description

This macro copies the given values buffer into the fifo and returns the number of copied elements.

kfifo_in_spinlocked_noirqsave

`kfifo_in_spinlocked_noirqsave (fifo, buf, n, lock)`

put data into fifo using a spinlock for locking, don't disable interrupts

Parameters

fifo
address of the fifo to be used

buf
the data to be added

n
number of elements to be added

lock
pointer to the spinlock to use for locking

Description

This is a variant of *kfifo_in_spinlocked()* but uses spin_lock/unlock() for locking and doesn't disable interrupts.

kfifo_out

`kfifo_out (fifo, buf, n)`

get data from the fifo

Parameters

fifo
address of the fifo to be used

buf
pointer to the storage buffer

n
max. number of elements to get

Description

This macro get some data from the fifo and return the numbers of elements copied.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_out_spinlocked

`kfifo_out_spinlocked (fifo, buf, n, lock)`

get data from the fifo using a spinlock for locking

Parameters

fifo
address of the fifo to be used

buf
pointer to the storage buffer

n
max. number of elements to get

lock
pointer to the spinlock to use for locking

Description

This macro get the data from the fifo and return the numbers of elements copied.

kfifo_out_spinlocked_noirqsave

`kfifo_out_spinlocked_noirqsave (fifo, buf, n, lock)`

get data from the fifo using a spinlock for locking, don't disable interrupts

Parameters

fifo
address of the fifo to be used

buf
pointer to the storage buffer

n
max. number of elements to get

lock
pointer to the spinlock to use for locking

Description

This is a variant of *kfifo_out_spinlocked()* which uses spin_lock/unlock() for locking and doesn't disable interrupts.

kfifo_from_user

`kfifo_from_user (fifo, from, len, copied)`

puts some data from user space into the fifo

Parameters

fifo
address of the fifo to be used

from
pointer to the data to be added

len
the length of the data to be added

copied
pointer to output variable to store the number of copied bytes

Description

This macro copies at most **len** bytes from the **from** into the fifo, depending of the available space and returns -EFAULT/0.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_to_user

`kfifo_to_user (fifo, to, len, copied)`

copies data from the fifo into user space

Parameters

fifo
address of the fifo to be used

to
where the data must be copied

len
the size of the destination buffer

copied
pointer to output variable to store the number of copied bytes

Description

This macro copies at most **len** bytes from the fifo into the **to** buffer and returns -EFAULT/0.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

kfifo_dma_in_prepare

`kfifo_dma_in_prepare (fifo, sgl, nents, len)`

setup a scatterlist for DMA input

Parameters

fifo
address of the fifo to be used

sgl
pointer to the scatterlist array

nents
number of entries in the scatterlist array

len
number of elements to transfer

Description

This macro fills a scatterlist for DMA input. It returns the number entries in the scatterlist array.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

kfifo_dma_in_finish

`kfifo_dma_in_finish (fifo, len)`

finish a DMA IN operation

Parameters

fifo
address of the fifo to be used

len
number of bytes to received

Description

This macro finish a DMA IN operation. The in counter will be updated by the len parameter. No error checking will be done.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

kfifo_dma_out_prepare

`kfifo_dma_out_prepare (fifo, sgl, nents, len)` setup a scatterlist for DMA output

Parameters

fifo
address of the fifo to be used

sgl
pointer to the scatterlist array

nents
number of entries in the scatterlist array

len
number of elements to transfer

Description

This macro fills a scatterlist for DMA output which at most **len** bytes to transfer. It returns the number entries in the scatterlist array. A zero means there is no space available and the scatterlist is not filled.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

kfifo_dma_out_finish

`kfifo_dma_out_finish (fifo, len)`

finish a DMA OUT operation

Parameters

fifo
address of the fifo to be used

len
number of bytes transferred

Description

This macro finish a DMA OUT operation. The out counter will be updated by the len parameter. No error checking will be done.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

kfifo_out_peek

`kfifo_out_peek (fifo, buf, n)`

gets some data from the fifo

Parameters

fifo
address of the fifo to be used

buf
pointer to the storage buffer

n
max. number of elements to get

Description

This macro get the data from the fifo and return the numbers of elements copied. The data is not removed from the fifo.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

1.1.7 relay interface support

Relay interface support is designed to provide an efficient mechanism for tools and facilities to relay large amounts of data from kernel space to user space.

relay interface

int **relay_buf_full**(struct rchan_buf *buf)

boolean, is the channel buffer full?

Parameters

struct rchan_buf *buf
channel buffer

Returns 1 if the buffer is full, 0 otherwise.

void **relay_reset**(struct rchan *chan)

reset the channel

Parameters

struct rchan *chan
the channel

This has the effect of erasing all data from all channel buffers and restarting the channel in its initial state. The buffers are not freed, so any mappings are still in effect.

NOTE. Care should be taken that the channel isn't actually being used by anything when this call is made.

struct rchan ***relay_open**(const char *base_filename, struct dentry *parent, size_t subbuf_size, size_t n_subbufs, const struct rchan_callbacks *cb, void *private_data)

create a new relay channel

Parameters

const char *base_filename
base name of files to create, NULL for buffering only

struct dentry *parent
dentry of parent directory, NULL for root directory or buffer

size_t subbuf_size
size of sub-buffers

size_t n_subbufs
number of sub-buffers

const struct rchan_callbacks *cb
client callback functions

void *private_data
user-defined data

Returns channel pointer if successful, NULL otherwise.

Creates a channel buffer for each cpu using the sizes and attributes specified. The created channel buffer files will be named base_filename0…base_filenameN-1. File permissions will be S_IRUSR.

If opening a buffer (**parent** = NULL) that you later wish to register in a filesystem, call relay_late_setup_files() once the **parent** dentry is available.

int **relay_late_setup_files**(struct rchan *chan, const char *base_filename, struct dentry *parent)

triggers file creation

Parameters

struct rchan *chan
channel to operate on

const char *base_filename
base name of files to create

struct dentry *parent
dentry of parent directory, NULL for root directory

Returns 0 if successful, non-zero otherwise.

Use to setup files for a previously buffer-only channel created by relay_open() with a NULL parent dentry.

For example, this is useful for perfomring early tracing in kernel, before VFS is up and then exposing the early results once the dentry is available.

size_t **relay_switch_subbuf**(struct rchan_buf *buf, size_t length)

switch to a new sub-buffer

Parameters

struct rchan_buf *buf
channel buffer

size_t length
size of current event

Returns either the length passed in or 0 if full.

Performs sub-buffer-switch tasks such as invoking callbacks, updating padding counts, waking up readers, etc.

void **relay_subbufs_consumed**(struct rchan *chan, unsigned int cpu, size_t subbufs_consumed)

update the buffer's sub-buffers-consumed count

Parameters

struct rchan *chan
the channel

unsigned int cpu
the cpu associated with the channel buffer to update

size_t subbufs_consumed
number of sub-buffers to add to current buf's count

Adds to the channel buffer's consumed sub-buffer count. subbufs_consumed should be the number of sub-buffers newly consumed, not the total consumed.

NOTE. Kernel clients don't need to call this function if the channel mode is 'overwrite'.

void **relay_close**(struct rchan *chan)
close the channel

Parameters

struct rchan *chan
the channel

Closes all channel buffers and frees the channel.

void **relay_flush**(struct rchan *chan)
close the channel

Parameters

struct rchan *chan
the channel

Flushes all channel buffers, i.e. forces buffer switch.

int **relay_mmap_buf**(struct rchan_buf *buf, struct vm_area_struct *vma)

- mmap channel buffer to process address space

Parameters

struct rchan_buf *buf
relay channel buffer

struct vm_area_struct *vma
vm_area_struct describing memory to be mapped

Returns 0 if ok, negative on error

Caller should already have grabbed mmap_lock.

void ***relay_alloc_buf**(struct rchan_buf *buf, size_t *size)
allocate a channel buffer

Parameters

struct rchan_buf *buf
the buffer struct

size_t *size
total size of the buffer

Returns a pointer to the resulting buffer, `NULL` if unsuccessful. The passed in size will get page aligned, if it isn't already.

struct rchan_buf ***relay_create_buf**(struct rchan *chan)
allocate and initialize a channel buffer

Parameters

struct rchan *chan
the relay channel

Returns channel buffer if successful, NULL otherwise.

void **relay_destroy_channel**(struct *kref* *kref)
free the channel struct

Parameters

struct kref *kref
target kernel reference that contains the relay channel

Should only be called from kref_put().

void **relay_destroy_buf**(struct rchan_buf *buf)
destroy an rchan_buf struct and associated buffer

Parameters

struct rchan_buf *buf
the buffer struct

void **relay_remove_buf**(struct *kref* *kref)
remove a channel buffer

Parameters

struct kref *kref
target kernel reference that contains the relay buffer

Removes the file from the filesystem, which also frees the rchan_buf_struct and the channel buffer. Should only be called from kref_put().

int **relay_buf_empty**(struct rchan_buf *buf)
boolean, is the channel buffer empty?

Parameters

struct rchan_buf *buf
channel buffer

Returns 1 if the buffer is empty, 0 otherwise.

void **wakeup_readers**(struct irq_work *work)
wake up readers waiting on a channel

Parameters

struct irq_work *work
contains the channel buffer

This is the function used to defer reader waking

void **__relay_reset**(struct rchan_buf *buf, unsigned int init)
reset a channel buffer

Parameters

struct rchan_buf *buf
the channel buffer

unsigned int init
1 if this is a first-time initialization

See relay_reset() for description of effect.

void **relay_close_buf**(struct rchan_buf *buf)
close a channel buffer

Parameters

struct rchan_buf *buf
channel buffer

Marks the buffer finalized and restores the default callbacks. The channel buffer and channel buffer data structure are then freed automatically when the last reference is given up.

int **relay_file_open**(struct *inode* *inode, struct file *filp)
open file op for relay files

Parameters

struct inode *inode
the inode

struct file *filp
the file

Increments the channel buffer refcount.

int **relay_file_mmap**(struct file *filp, struct vm_area_struct *vma)
mmap file op for relay files

Parameters

struct file *filp
the file

struct vm_area_struct *vma
the vma describing what to map

Calls upon *relay_mmap_buf()* to map the file into user space.

__poll_t **relay_file_poll**(struct file *filp, poll_table *wait)
poll file op for relay files

Parameters

struct file *filp
the file

poll_table *wait
poll table

Poll implemention.

int **relay_file_release**(struct *inode* *inode, struct file *filp)

release file op for relay files

Parameters

struct inode *inode
the inode

struct file *filp
the file

Decrements the channel refcount, as the filesystem is no longer using it.

size_t **relay_file_read_subbuf_avail**(size_t read_pos, struct rchan_buf *buf)

return bytes available in sub-buffer

Parameters

size_t read_pos
file read position

struct rchan_buf *buf
relay channel buffer

size_t **relay_file_read_start_pos**(struct rchan_buf *buf)

find the first available byte to read

Parameters

struct rchan_buf *buf
relay channel buffer

If the read_pos is in the middle of padding, return the position of the first actually available byte, otherwise return the original value.

size_t **relay_file_read_end_pos**(struct rchan_buf *buf, size_t read_pos, size_t count)

return the new read position

Parameters

struct rchan_buf *buf
relay channel buffer

size_t read_pos
file read position

size_t count
number of bytes to be read

1.1.8 Module Support

Kernel module auto-loading

int **__request_module**(bool wait, const char *fmt, ...)
try to load a kernel module

Parameters

bool wait
wait (or not) for the operation to complete

const char *fmt
printf style format string for the name of the module

...
arguments as specified in the format string

Description

Load a module using the user mode module loader. The function returns zero on success or a negative errno code or positive exit code from "modprobe" on failure. Note that a successful module load does not mean the module did not then unload and exit on an error of its own. Callers must check that the service they requested is now available not blindly invoke it.

If module auto-loading support is disabled then this function simply returns -ENOENT.

Module debugging

Enabling CONFIG_MODULE_STATS enables module debugging statistics which are useful to monitor and root cause memory pressure issues with module loading. These statistics are useful to allow us to improve production workloads.

The current module debugging statistics supported help keep track of module loading failures to enable improvements either for kernel module auto-loading usage (request_module()) or interactions with userspace. Statistics are provided to track all possible failures in the finit_module() path and memory wasted in this process space. Each of the failure counters are associated to a type of module loading failure which is known to incur a certain amount of memory allocation loss. In the worst case loading a module will fail after a 3 step memory allocation process:

a) memory allocated with kernel_read_file_from_fd()

b) module decompression processes the file read from kernel_read_file_from_fd(), and vmap() is used to map the decompressed module to a new local buffer which represents a copy of the decompressed module passed from userspace. The buffer from kernel_read_file_from_fd() is freed right away.

c) layout_and_allocate() allocates space for the final resting place where we would keep the module if it were to be processed successfully.

If a failure occurs after these three different allocations only one counter will be incremented with the summation of the allocated bytes freed incurred during this failure. Likewise, if module loading failed only after step b) a separate counter is used and incremented for the bytes freed and not used during both of those allocations.

Virtual memory space can be limited, for example on x86 virtual memory size defaults to 128 MiB. We should strive to limit and avoid wasting virtual memory allocations when possible.

These module debugging statistics help to evaluate how much memory is being wasted on bootup due to module loading failures.

All counters are designed to be incremental. Atomic counters are used so to remain simple and avoid delays and deadlocks.

dup_failed_modules - tracks duplicate failed modules

Linked list of modules which failed to be loaded because an already existing module with the same name was already being processed or already loaded. The finit_module() system call incurs heavy virtual memory allocations. In the worst case an finit_module() system call can end up allocating virtual memory 3 times:

1) kernel_read_file_from_fd() call uses vmalloc()
2) optional module decompression uses vmap()
3) layout_and allocate() can use vzalloc() or an arch specific variation of vmalloc to deal with ELF sections requiring special permissions

In practice on a typical boot today most finit_module() calls fail due to the module with the same name already being loaded or about to be processed. All virtual memory allocated to these failed modules will be freed with no functional use.

To help with this the dup_failed_modules allows us to track modules which failed to load due to the fact that a module was already loaded or being processed. There are only two points at which we can fail such calls, we list them below along with the number of virtual memory allocation calls:

a) FAIL_DUP_MOD_BECOMING: at the end of early_mod_check() before layout_and_allocate(). - with module decompression: 2 virtual memory allocation calls - without module decompression: 1 virtual memory allocation calls
b) FAIL_DUP_MOD_LOAD: after layout_and_allocate() on add_unformed_module() - with module decompression 3 virtual memory allocation calls - without module decompression 2 virtual memory allocation calls

We should strive to get this list to be as small as possible. If this list is not empty it is a reflection of possible work or optimizations possible either in-kernel or in userspace.

module statistics debugfs counters

The total amount of wasted virtual memory allocation space during module loading can be computed by adding the total from the summation:

- **invalid_kread_bytes + invalid_decompress_bytes + invalid_becoming_bytes + invalid_mod_bytes**

The following debugfs counters are available to inspect module loading failures:

- total_mod_size: total bytes ever used by all modules we've dealt with on this system
- total_text_size: total bytes of the .text and .init.text ELF section sizes we've dealt with on this system

- invalid_kread_bytes: bytes allocated and then freed on failures which happen due to the initial kernel_read_file_from_fd(). kernel_read_file_from_fd() uses vmalloc(). These should typically not happen unless your system is under memory pressure.
- invalid_decompress_bytes: number of bytes allocated and freed due to memory allocations in the module decompression path that use vmap(). These typically should not happen unless your system is under memory pressure.
- invalid_becoming_bytes: total number of bytes allocated and freed used used to read the kernel module userspace wants us to read before we promote it to be processed to be added to our **modules** linked list. These failures can happen if we had a check in between a successful kernel_read_file_from_fd() call and right before we allocate the our private memory for the module which would be kept if the module is successfully loaded. The most common reason for this failure is when userspace is racing to load a module which it does not yet see loaded. The first module to succeed in add_unformed_module() will add a module to our `modules` list and subsequent loads of modules with the same name will error out at the end of early_mod_check(). The check for module_patient_check_exists() at the end of early_mod_check() prevents duplicate allocations on layout_and_allocate() for modules already being processed. These duplicate failed modules are non-fatal, however they typically are indicative of userspace not seeing a module in userspace loaded yet and unnecessarily trying to load a module before the kernel even has a chance to begin to process prior requests. Although duplicate failures can be non-fatal, we should try to reduce vmalloc() pressure proactively, so ideally after boot this will be close to as 0 as possible. If module decompression was used we also add to this counter the cost of the initial kernel_read_file_from_fd() of the compressed module. If module decompression was not used the value represents the total allocated and freed bytes in kernel_read_file_from_fd() calls for these type of failures. These failures can occur because:
- module_sig_check() - module signature checks
- elf_validity_cache_copy() - some ELF validation issue
- early_mod_check():
  - blacklisting
  - failed to rewrite section headers
  - version magic
  - live patch requirements didn't check out
  - the module was detected as being already present
- invalid_mod_bytes: these are the total number of bytes allocated and freed due to failures after we did all the sanity checks of the module which userspace passed to us and after our first check that the module is unique. A module can still fail to load if we detect the module is loaded after we allocate space for it with layout_and_allocate(), we do

this check right before processing the module as live and run its initialization routines. Note that you have a failure of this type it also means the respective kernel_read_file_from_fd() memory space was also freed and not used, and so we increment this counter with twice the size of the module. Additionally if you used module decompression the size of the compressed module is also added to this counter.

- modcount: how many modules we've loaded in our kernel life time
- failed_kreads: how many modules failed due to failed kernel_read_file_from_fd()
- failed_decompress: how many failed module decompression attempts we've had. These really should not happen unless your compression / decompression might be broken.
- failed_becoming: how many modules failed after we kernel_read_file_from_fd() it and before we allocate memory for it with layout_and_allocate(). This counter is never incremented if you manage to validate the module and call layout_and_allocate() for it.
- failed_load_modules: how many modules failed once we've allocated our private space for our module using layout_and_allocate(). These failures should hopefully mostly be dealt with already. Races in theory could still exist here, but it would just mean the kernel had started processing two threads concurrently up to early_mod_check() and one thread won. These failures are good signs the kernel or userspace is doing something seriously stupid or that could be improved. We should strive to fix these, but it is perhaps not easy to fix them. A recent example are the modules requests incurred for frequency modules, a separate module request was being issued for each CPU on a system.

Inter Module support

Refer to the files in kernel/module/ for more information.

1.1.9 Hardware Interfaces

DMA Channels

int **request_dma**(unsigned int dmanr, const char *device_id)
request and reserve a system DMA channel

Parameters

unsigned int dmanr
DMA channel number

const char * device_id
reserving device ID string, used in /proc/dma

void **free_dma**(unsigned int dmanr)
free a reserved system DMA channel

Parameters

unsigned int dmanr
DMA channel number

Resources Management

struct resource ***request_resource_conflict**(struct resource *root, struct resource *new)
request and reserve an I/O or memory resource

Parameters

struct resource *root
root resource descriptor

struct resource *new
resource descriptor desired by caller

Description

Returns 0 for success, conflict resource on error.

int **find_next_iomem_res**(resource_size_t start, resource_size_t end, unsigned long flags, unsigned long desc, struct resource *res)
Finds the lowest iomem resource that covers part of [**start**..**end**].

Parameters

resource_size_t start
start address of the resource searched for

resource_size_t end
end address of same resource

unsigned long flags
flags which the resource must have

unsigned long desc
descriptor the resource must have

struct resource *res
return ptr, if resource found

Description

If a resource is found, returns 0 and ***res is overwritten with the part of the resource that's within [**start**..**end**]; if none is found, returns -ENODEV. Returns -EINVAL for invalid parameters.

The caller must specify **start**, **end**, **flags**, and **desc** (which may be IORES_DESC_NONE).

int **reallocate_resource**(struct resource *root, struct resource *old, resource_size_t newsize, struct resource_constraint *constraint)
allocate a slot in the resource tree given range & alignment. The resource will be relocated if the new size cannot be reallocated in the current location.

Parameters

struct resource *root
root resource descriptor

struct resource *old
resource descriptor desired by caller

resource_size_t newsize
new size of the resource descriptor

struct resource_constraint *constraint
the size and alignment constraints to be met.

struct resource ***lookup_resource**(struct resource *root, resource_size_t start)
find an existing resource by a resource start address

Parameters

struct resource *root
root resource descriptor

resource_size_t start
resource start address

Description

Returns a pointer to the resource if found, NULL otherwise

struct resource ***insert_resource_conflict**(struct resource *parent, struct resource *new)
Inserts resource in the resource tree

Parameters

struct resource *parent
parent of the new resource

struct resource *new
new resource to insert

Description

Returns 0 on success, conflict resource if the resource can't be inserted.

This function is equivalent to request_resource_conflict when no conflict happens. If a conflict happens, and the conflicting resources entirely fit within the range of the new resource, then the new resource is inserted and the conflicting resources become children of the new resource.

This function is intended for producers of resources, such as FW modules and bus drivers.

resource_size_t **resource_alignment**(struct resource *res)
calculate resource's alignment

Parameters

struct resource *res
resource pointer

Description

Returns alignment on success, 0 (invalid alignment) on failure.

void **release_mem_region_adjustable**(resource_size_t start, resource_size_t size)
release a previously reserved memory region

Parameters

resource_size_t start
resource start address

resource_size_t size
resource region size

Description

This interface is intended for memory hot-delete. The requested region is released from a currently busy memory resource. The requested region must either match exactly or fit into a single busy resource entry. In the latter case, the remaining resource is adjusted accordingly. Existing children of the busy memory resource must be immutable in the request.

Note

- Additional release conditions, such as overlapping region, can be supported after they are confirmed as valid cases.
- When a busy memory resource gets split into two entries, the code assumes that all children remain in the lower address entry for simplicity. Enhance this logic when necessary.

void **merge_system_ram_resource**(struct resource *res)
mark the System RAM resource mergeable and try to merge it with adjacent, mergeable resources

Parameters

struct resource *res
resource descriptor

Description

This interface is intended for memory hotplug, whereby lots of contiguous system ram resources are added (e.g., via add_memory*()) by a driver, and the actual resource boundaries are not of interest (e.g., it might be relevant for DIMMs). Only resources that are marked mergeable, that have the same parent, and that don't have any children are considered. All mergeable resources must be immutable during the request.

Note

- The caller has to make sure that no pointers to resources that are marked mergeable are used anymore after this call - the resource might be freed and the pointer might be stale!
- *release_mem_region_adjustable()* will split on demand on memory hotunplug

int **request_resource**(struct resource *root, struct resource *new)
request and reserve an I/O or memory resource

Parameters

struct resource *root
root resource descriptor

struct resource *new
resource descriptor desired by caller

Description

Returns 0 for success, negative error code on error.

int **release_resource**(struct resource *old)
release a previously reserved resource

Parameters

struct resource *old
resource pointer

int **walk_iomem_res_desc**(unsigned long desc, unsigned long flags, u64 start, u64 end, void *arg, int (*func)(struct resource*, void*))
Walks through iomem resources and calls func() with matching resource ranges. *

Parameters

unsigned long desc
I/O resource descriptor. Use IORES_DESC_NONE to skip **desc** check.

unsigned long flags
I/O resource flags

u64 start
start addr

u64 end
end addr

void *arg
function argument for the callback **func**

int (*func)(struct resource *, void *)
callback function that is called for each qualifying resource area

Description

All the memory ranges which overlap start,end and also match flags and desc are valid candidates.

NOTE

For a new descriptor search, define a new IORES_DESC in <linux/ioport.h> and set it in 'desc' of a target resource entry.

int **region_intersects**(resource_size_t start, size_t size, unsigned long flags, unsigned long desc)
determine intersection of region with known resources

Parameters

resource_size_t start
region start address

size_t size
size of region

unsigned long flags
flags of resource (in iomem_resource)

unsigned long desc
descriptor of resource (in iomem_resource) or IORES_DESC_NONE

Description

Check if the specified region partially overlaps or fully eclipses a resource identified by **flags** and **desc** (optional with IORES_DESC_NONE). Return REGION_DISJOINT if the region does not overlap **flags/desc**, return REGION_MIXED if the region overlaps **flags/desc** and another resource, and return REGION_INTERSECTS if the region overlaps **flags/desc** and no other defined resource. Note that REGION_INTERSECTS is also returned in the case when the specified region overlaps RAM and undefined memory holes.

region_intersect() is used by memory remapping functions to ensure the user is not remapping RAM and is a vast speed up over walking through the resource table page by page.

int **allocate_resource**(struct resource *root, struct resource *new, resource_size_t size, resource_size_t min, resource_size_t max, resource_size_t align, resource_size_t (*alignf)(void*, const struct resource*, resource_size_t, resource_size_t), void *alignf_data)

allocate empty slot in the resource tree given range & alignment. The resource will be reallocated with a new size if it was already allocated

Parameters

struct resource *root
root resource descriptor

struct resource *new
resource descriptor desired by caller

resource_size_t size
requested resource region size

resource_size_t min
minimum boundary to allocate

resource_size_t max
maximum boundary to allocate

resource_size_t align
alignment requested, in bytes

resource_size_t (*alignf)(void *, const struct resource *, resource_size_t, resource_size_t)
alignment function, optional, called if not NULL

void *alignf_data
arbitrary data to pass to the **alignf** function

int **insert_resource**(struct resource *parent, struct resource *new)

Inserts a resource in the resource tree

Parameters

struct resource *parent
parent of the new resource

struct resource *new
new resource to insert

Description

Returns 0 on success, -EBUSY if the resource can't be inserted.

This function is intended for producers of resources, such as FW modules and bus drivers.

void **insert_resource_expand_to_fit**(struct resource *root, struct resource *new)
Insert a resource into the resource tree

Parameters

struct resource *root
root resource descriptor

struct resource *new
new resource to insert

Description

Insert a resource into the resource tree, possibly expanding it in order to make it encompass any conflicting resources.

int **remove_resource**(struct resource *old)
Remove a resource in the resource tree

Parameters

struct resource *old
resource to remove

Description

Returns 0 on success, -EINVAL if the resource is not valid.

This function removes a resource previously inserted by *insert_resource()* or *insert_resource_conflict()*, and moves the children (if any) up to where they were before. *insert_resource()* and *insert_resource_conflict()* insert a new resource, and move any conflicting resources down to the children of the new resource.

insert_resource(), *insert_resource_conflict()* and *remove_resource()* are intended for producers of resources, such as FW modules and bus drivers.

int **adjust_resource**(struct resource *res, resource_size_t start, resource_size_t size)
modify a resource's start and size

Parameters

struct resource *res
resource to modify

resource_size_t start
new start value

resource_size_t size
new size

Description

Given an existing resource, change its start and size to match the arguments. Returns 0 on success, -EBUSY if it can't fit. Existing children of the resource are assumed to be immutable.

struct resource ***__request_region**(struct resource *parent, resource_size_t start, resource_size_t n, const char *name, int flags)
create a new busy resource region

Parameters

struct resource *parent
parent resource descriptor

resource_size_t start
resource start address

resource_size_t n
resource region size

const char *name
reserving caller's ID string

int flags
IO resource flags

void **__release_region**(struct resource *parent, resource_size_t start, resource_size_t n)
release a previously reserved resource region

Parameters

struct resource *parent
parent resource descriptor

resource_size_t start
resource start address

resource_size_t n
resource region size

Description

The described resource region must match a currently busy region.

int **devm_request_resource**(struct device *dev, struct resource *root, struct resource *new)
request and reserve an I/O or memory resource

Parameters

struct device *dev
device for which to request the resource

struct resource *root
root of the resource tree from which to request the resource

struct resource *new
descriptor of the resource to request

Description

This is a device-managed version of *request_resource()*. There is usually no need to release resources requested by this function explicitly since that will be taken care of when the device is unbound from its driver. If for some reason the resource needs to be released explicitly, because of ordering issues for example, drivers must call devm_release_resource() rather than the regular *release_resource()*.

When a conflict is detected between any existing resources and the newly requested resource, an error message will be printed.

Returns 0 on success or a negative error code on failure.

void **devm_release_resource**(struct device *dev, struct resource *new)

release a previously requested resource

Parameters

struct device *dev
device for which to release the resource

struct resource *new
descriptor of the resource to release

Description

Releases a resource previously requested using devm_request_resource().

struct resource ***devm_request_free_mem_region**(struct device *dev, struct resource *base, unsigned long size)

find free region for device private memory

Parameters

struct device *dev
device struct to bind the resource to

struct resource *base
resource tree to look in

unsigned long size
size in bytes of the device memory to add

Description

This function tries to find an empty range of physical address big enough to contain the new resource, so that it can later be hotplugged as ZONE_DEVICE memory, which in turn allocates struct pages.

struct resource ***alloc_free_mem_region**(struct resource *base, unsigned long size, unsigned long align, const char *name)

find a free region relative to **base**

Parameters

struct resource *base
resource that will parent the new resource

unsigned long size
size in bytes of memory to allocate from **base**

unsigned long align
alignment requirements for the allocation

const char *name
resource name

Description

Buses like CXL, that can dynamically instantiate new memory regions, need a method to allocate physical address space for those regions. Allocate and insert a new resource to cover a free, unclaimed by a descendant of **base**, range in the span of **base**.

MTRR Handling

int **arch_phys_wc_add**(unsigned long base, unsigned long size)
add a WC MTRR and handle errors if PAT is unavailable

Parameters

unsigned long base
Physical base address

unsigned long size
Size of region

Description

If PAT is available, this does nothing. If PAT is unavailable, it attempts to add a WC MTRR covering size bytes starting at base and logs an error if this fails.

The called should provide a power of two size on an equivalent power of two boundary.

Drivers must store the return value to pass to mtrr_del_wc_if_needed, but drivers should not try to interpret that return value.

1.1.10 Security Framework

int **security_init**(void)
initializes the security framework

Parameters

void
no arguments

Description

This should be called early in the kernel initialization sequence.

void **security_add_hooks**(struct security_hook_list *hooks, int count, const char *lsm)
Add a modules hooks to the hook lists.

Parameters

struct security_hook_list *hooks
the hooks to add

int count
the number of hooks to add

const char *lsm
the name of the security module

Description

Each LSM has to register its hooks with the infrastructure.

int **lsm_cred_alloc**(struct *cred* *cred, gfp_t gfp)
allocate a composite cred blob

Parameters

struct cred *cred
the cred that needs a blob

gfp_t gfp
allocation type

Description

Allocate the cred blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

void **lsm_early_cred**(struct *cred* *cred)
during initialization allocate a composite cred blob

Parameters

struct cred *cred
the cred that needs a blob

Description

Allocate the cred blob for all the modules

int **lsm_file_alloc**(struct *file* *file)
allocate a composite file blob

Parameters

struct file *file
the file that needs a blob

Description

Allocate the file blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

int **lsm_inode_alloc**(struct *inode* *inode)
allocate a composite inode blob

Parameters

struct inode *inode
the inode that needs a blob

Description

Allocate the inode blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

int **lsm_task_alloc**(struct task_struct *task)
allocate a composite task blob

Parameters

struct task_struct *task
the task that needs a blob

Description

Allocate the task blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

int **lsm_ipc_alloc**(struct kern_ipc_perm *kip)
allocate a composite ipc blob

Parameters

struct kern_ipc_perm *kip
the ipc that needs a blob

Description

Allocate the ipc blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

int **lsm_msg_msg_alloc**(struct msg_msg *mp)
allocate a composite msg_msg blob

Parameters

struct msg_msg *mp
the msg_msg that needs a blob

Description

Allocate the ipc blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

void **lsm_early_task**(struct task_struct *task)
during initialization allocate a composite task blob

Parameters

struct task_struct *task
the task that needs a blob

Description

Allocate the task blob for all the modules

int **lsm_superblock_alloc**(struct super_block *sb)
allocate a composite superblock blob

Parameters

struct super_block *sb
the superblock that needs a blob

Description

Allocate the superblock blob for all the modules

Returns 0, or -ENOMEM if memory can't be allocated.

int **security_binder_set_context_mgr**(const struct cred *mgr)
Check if becoming binder ctx mgr is ok

Parameters

const struct cred *mgr
task credentials of current binder process

Description

Check whether **mgr** is allowed to be the binder context manager.

Return

Return 0 if permission is granted.

int **security_binder_transaction**(const struct cred *from, const struct cred *to)

Check if a binder transaction is allowed

Parameters

const struct cred *from
sending process

const struct cred *to
receiving process

Description

Check whether **from** is allowed to invoke a binder transaction call to **to**.

Return

Returns 0 if permission is granted.

int **security_binder_transfer_binder**(const struct cred *from, const struct cred *to)

Check if a binder transfer is allowed

Parameters

const struct cred *from
sending process

const struct cred *to
receiving process

Description

Check whether **from** is allowed to transfer a binder reference to **to**.

Return

Returns 0 if permission is granted.

int **security_binder_transfer_file**(const struct cred *from, const struct cred *to, const struct *file* *file)

Check if a binder file xfer is allowed

Parameters

const struct cred *from
sending process

const struct cred *to
receiving process

const struct file *file
file being transferred

Description

Check whether **from** is allowed to transfer **file** to **to**.

Return

Returns 0 if permission is granted.

int **security_ptrace_access_check**(struct task_struct *child, unsigned int mode)

Check if tracing is allowed

Parameters

struct task_struct *child
target process

unsigned int mode
PTRACE_MODE flags

Description

Check permission before allowing the current process to trace the **child** process. Security modules may also want to perform a process tracing check during an execve in the set_security or apply_creds hooks of tracing check during an execve in the bprm_set_creds hook of binprm_security_ops if the process is being traced and its security attributes would be changed by the execve.

Return

Returns 0 if permission is granted.

int **security_ptrace_traceme**(struct task_struct *parent)

Check if tracing is allowed

Parameters

struct task_struct *parent
tracing process

Description

Check that the **parent** process has sufficient permission to trace the current process before allowing the current process to present itself to the **parent** process for tracing.

Return

Returns 0 if permission is granted.

int **security_capget**(const struct task_struct *target, kernel_cap_t *effective, kernel_cap_t *inheritable, kernel_cap_t *permitted)

Get the capability sets for a process

Parameters

const struct task_struct *target
target process

kernel_cap_t *effective
effective capability set

kernel_cap_t *inheritable
inheritable capability set

kernel_cap_t *permitted
permitted capability set

Description

Get the **effective**, **inheritable**, and **permitted** capability sets for the **target** process. The hook may also perform permission checking to determine if the current process is allowed to see the capability sets of the **target** process.

Return

Returns 0 if the capability sets were successfully obtained.

int **security_capset**(struct cred *new, const struct cred *old, const kernel_cap_t *effective, const kernel_cap_t *inheritable, const kernel_cap_t *permitted)

Set the capability sets for a process

Parameters

struct cred *new
new credentials for the target process

const struct cred *old
current credentials of the target process

const kernel_cap_t *effective
effective capability set

const kernel_cap_t *inheritable
inheritable capability set

const kernel_cap_t *permitted
permitted capability set

Description

Set the **effective**, **inheritable**, and **permitted** capability sets for the current process.

Return

Returns 0 and update **new** if permission is granted.

int **security_capable**(const struct *cred* *cred, struct user_namespace *ns, int cap, unsigned int opts)

Check if a process has the necessary capability

Parameters

const struct cred *cred
credentials to examine

struct user_namespace *ns
user namespace

int cap
capability requested

unsigned int opts
capability check options

Description

Check whether the **tsk** process has the **cap** capability in the indicated credentials. **cap** contains the capability <include/linux/capability.h>. **opts** contains options for the capable check <include/linux/security.h>.

Return

Returns 0 if the capability is granted.

int **security_quotactl**(int cmds, int type, int id, struct super_block *sb)

Check if a quotactl() syscall is allowed for this fs

Parameters

int cmds
commands

int type
type

int id
id

struct super_block *sb
filesystem

Description

Check whether the quotactl syscall is allowed for this **sb**.

Return

Returns 0 if permission is granted.

int **security_quota_on**(struct *dentry* *dentry)

Check if QUOTAON is allowed for a dentry

Parameters

struct dentry *dentry
dentry

Description

Check whether QUOTAON is allowed for **dentry**.

Return

Returns 0 if permission is granted.

int **security_syslog**(int type)

Check if accessing the kernel message ring is allowed

Parameters

int type
SYSLOG_ACTION_* type

Description

Check permission before accessing the kernel message ring or changing logging to the console. See the syslog(2) manual page for an explanation of the **type** values.

Return

Return 0 if permission is granted.

int **security_settime64**(const struct timespec64 *ts, const struct timezone *tz)

Check if changing the system time is allowed

Parameters

const struct timespec64 *ts
new time

const struct timezone *tz
timezone

Description

Check permission to change the system time, struct timespec64 is defined in <include/linux/time64.h> and timezone is defined in <include/linux/time.h>.

Return

Returns 0 if permission is granted.

int **security_vm_enough_memory_mm**(struct mm_struct *mm, long pages)

Check if allocating a new mem map is allowed

Parameters

struct mm_struct *mm
mm struct

long pages
number of pages

Description

Check permissions for allocating a new virtual mapping. If all LSMs return a positive value, __vm_enough_memory() will be called with cap_sys_admin set. If at least one LSM returns 0 or negative, __vm_enough_memory() will be called with cap_sys_admin cleared.

Return

Returns 0 if permission is granted by the LSM infrastructure to the
caller.

int **security_bprm_creds_for_exec**(struct linux_binprm *bprm)

Prepare the credentials for exec()

Parameters

struct linux_binprm *bprm
binary program information

Description

If the setup in prepare_exec_creds did not setup **bprm->cred->security** properly for executing **bprm->file**, update the LSM's portion of **bprm->cred->security** to be what commit_creds needs to install for the new program. This hook may also optionally check permissions (e.g. for transitions between security domains). The hook must set **bprm->secureexec** to 1 if AT_SECURE should be set to request libc enable secure mode. **bprm** contains the linux_binprm structure.

Return

Returns 0 if the hook is successful and permission is granted.

int **security_bprm_creds_from_file**(struct linux_binprm *bprm, struct *file* *file)

Update linux_binprm creds based on file

Parameters

struct linux_binprm *bprm
binary program information

struct file *file
associated file

Description

If **file** is setpcap, suid, sgid or otherwise marked to change privilege upon exec, update **bprm->cred** to reflect that change. This is called after finding the binary that will be executed without an interpreter. This ensures that the credentials will not be derived from a script that the binary will need to reopen, which when reopend may end up being a completely different file. This hook may also optionally check permissions (e.g. for transitions between security domains). The hook must set **bprm->secureexec** to 1 if AT_SECURE should be set to request libc enable secure mode. The hook must add to **bprm->per_clear** any personality flags that should be cleared from current->personality. **bprm** contains the linux_binprm structure.

Return

Returns 0 if the hook is successful and permission is granted.

int **security_bprm_check**(struct linux_binprm *bprm)

Mediate binary handler search

Parameters

struct linux_binprm *bprm
binary program information

Description

This hook mediates the point when a search for a binary handler will begin. It allows a check against the **bprm->cred->security** value which was set in the preceding creds_for_exec call. The argv list and envp list are reliably available in **bprm**. This hook may be called multiple times during a single execve. **bprm** contains the linux_binprm structure.

Return

Returns 0 if the hook is successful and permission is granted.

void **security_bprm_committing_creds**(struct linux_binprm *bprm)

Install creds for a process during exec()

Parameters

struct linux_binprm *bprm
binary program information

Description

Prepare to install the new security attributes of a process being transformed by an execve operation, based on the old credentials pointed to by **current->cred** and the information set in **bprm->cred** by the bprm_creds_for_exec hook. **bprm** points to the linux_binprm structure. This hook is a good place to perform state changes on the process such as closing open file descriptors to which access will no longer be granted when the attributes are changed. This is called immediately before commit_creds().

void **security_bprm_committed_creds**(struct linux_binprm *bprm)

Tidy up after cred install during exec()

Parameters

struct linux_binprm *bprm
binary program information

Description

Tidy up after the installation of the new security attributes of a process being transformed by an execve operation. The new credentials have, by this point, been set to **current->cred**. **bprm** points to the linux_binprm structure. This hook is a good place to perform state changes on the process such as clearing out non-inheritable signal state. This is called immediately after commit_creds().

int **security_fs_context_submount**(struct fs_context *fc, struct super_block *reference)

Initialise fc->security

Parameters

struct fs_context *fc
new filesystem context

struct super_block *reference
dentry reference for submount/remount

Description

Fill out the ->security field for a new fs_context.

Return

Returns 0 on success or negative error code on failure.

int **security_fs_context_dup**(struct fs_context *fc, struct fs_context *src_fc)

Duplicate a fs_context LSM blob

Parameters

struct fs_context *fc
destination filesystem context

struct fs_context *src_fc
source filesystem context

Description

Allocate and attach a security structure to sc->security. This pointer is initialised to NULL by the caller. **fc** indicates the new filesystem context. **src_fc** indicates the original filesystem context.

Return

Returns 0 on success or a negative error code on failure.

int **security_fs_context_parse_param**(struct fs_context *fc, struct fs_parameter *param)
Configure a filesystem context

Parameters

struct fs_context *fc
filesystem context

struct fs_parameter *param
filesystem parameter

Description

Userspace provided a parameter to configure a superblock. The LSM can consume the parameter or return it to the caller for use elsewhere.

Return

If the parameter is used by the LSM it should return 0, if it is
returned to the caller -ENOPARAM is returned, otherwise a negative error code is returned.

int **security_sb_alloc**(struct super_block *sb)
Allocate a super_block LSM blob

Parameters

struct super_block *sb
filesystem superblock

Description

Allocate and attach a security structure to the sb->s_security field. The s_security field is initialized to NULL when the structure is allocated. **sb** contains the super_block structure to be modified.

Return

Returns 0 if operation was successful.

void **security_sb_delete**(struct super_block *sb)
Release super_block LSM associated objects

Parameters

struct super_block *sb
filesystem superblock

Description

Release objects tied to a superblock (e.g. inodes). **sb** contains the super_block structure being released.

void **security_sb_free**(struct super_block *sb)
Free a super_block LSM blob

Parameters

struct super_block *sb
filesystem superblock

Description

Deallocate and clear the sb->s_security field. **sb** contains the super_block structure to be modified.

int **security_sb_kern_mount**(struct super_block *sb)
Check if a kernel mount is allowed

Parameters

struct super_block *sb
filesystem superblock

Description

Mount this **sb** if allowed by permissions.

Return

Returns 0 if permission is granted.

int **security_sb_show_options**(struct seq_file *m, struct super_block *sb)
Output the mount options for a superblock

Parameters

struct seq_file *m
output file

struct super_block *sb
filesystem superblock

Description

Show (print on **m**) mount options for this **sb**.

Return

Returns 0 on success, negative values on failure.

int **security_sb_statfs**(struct *dentry* *dentry)
Check if accessing fs stats is allowed

Parameters

struct dentry *dentry
superblock handle

Description

Check permission before obtaining filesystem statistics for the **mnt** mountpoint. **dentry** is a handle on the superblock for the filesystem.

Return

Returns 0 if permission is granted.

int **security_sb_mount**(const char *dev_name, const struct *path* *path, const char *type, unsigned long flags, void *data)
Check permission for mounting a filesystem

Parameters

const char *dev_name
filesystem backing device

const struct path *path
mount point

const char *type
filesystem type

unsigned long flags
mount flags

void *data
filesystem specific data

Description

Check permission before an object specified by **dev_name** is mounted on the mount point named by **nd**. For an ordinary mount, **dev_name** identifies a device if the file system type requires a device. For a remount (**flags** & MS_REMOUNT), **dev_name** is irrelevant. For a loopback/bind mount (**flags** & MS_BIND), **dev_name** identifies the pathname of the object being mounted.

Return

Returns 0 if permission is granted.

int **security_sb_umount**(struct vfsmount *mnt, int flags)
Check permission for unmounting a filesystem

Parameters

struct vfsmount *mnt
mounted filesystem

int flags
unmount flags

Description

Check permission before the **mnt** file system is unmounted.

Return

Returns 0 if permission is granted.

int **security_sb_pivotroot**(const struct path *old_path, const struct path *new_path)
Check permissions for pivoting the rootfs

Parameters

const struct path *old_path
new location for current rootfs

const struct path *new_path
location of the new rootfs

Description

Check permission before pivoting the root filesystem.

Return

Returns 0 if permission is granted.

int **security_move_mount**(const struct path *from_path, const struct path *to_path)
Check permissions for moving a mount

Parameters

const struct path *from_path
source mount point

const struct path *to_path
destination mount point

Description

Check permission before a mount is moved.

Return

Returns 0 if permission is granted.

int **security_path_notify**(const struct *path* *path, u64 mask, unsigned int obj_type)
Check if setting a watch is allowed

Parameters

const struct path *path
file path

u64 mask
event mask

unsigned int obj_type
file path type

Description

Check permissions before setting a watch on events as defined by **mask**, on an object at **path**, whose type is defined by **obj_type**.

Return

Returns 0 if permission is granted.

int **security_inode_alloc**(struct *inode* *inode)
Allocate an inode LSM blob

Parameters

struct inode *inode
the inode

Description

Allocate and attach a security structure to **inode->i_security**. The i_security field is initialized to NULL when the inode structure is allocated.

Return

Return 0 if operation was successful.

void **security_inode_free**(struct *inode* *inode)

Free an inode's LSM blob

Parameters

struct inode *inode
the inode

Description

Deallocate the inode security structure and set **inode->i_security** to NULL.

int **security_inode_init_security_anon**(struct *inode* *inode, const struct qstr *name, const struct *inode* *context_inode)

Initialize an anonymous inode

Parameters

struct inode *inode
the inode

const struct qstr *name
the anonymous inode class

const struct inode *context_inode
an optional related inode

Description

Set up the incore security field for the new anonymous inode and return whether the inode creation is permitted by the security module or not.

Return

Returns 0 on success, -EACCES if the security module denies the creation of this inode, or another -errno upon other errors.

int **security_path_rmdir**(const struct path *dir, struct *dentry* *dentry)

Check if removing a directory is allowed

Parameters

const struct path *dir
parent directory

struct dentry *dentry
directory to remove

Description

Check the permission to remove a directory.

Return

Returns 0 if permission is granted.

int **security_path_symlink**(const struct path *dir, struct *dentry* *dentry, const char *old_name)

Check if creating a symbolic link is allowed

Parameters

const struct path *dir
    parent directory

struct dentry *dentry
    symbolic link

const char *old_name
    file pathname

Description

Check the permission to create a symbolic link to a file.

Return

Returns 0 if permission is granted.

int **security_path_link**(struct dentry *old_dentry, const struct path *new_dir, struct dentry *new_dentry)
    Check if creating a hard link is allowed

Parameters

struct dentry *old_dentry
    existing file

const struct path *new_dir
    new parent directory

struct dentry *new_dentry
    new link

Description

Check permission before creating a new hard link to a file.

Return

Returns 0 if permission is granted.

int **security_path_truncate**(const struct *path* *path)
    Check if truncating a file is allowed

Parameters

const struct path *path
    file

Description

Check permission before truncating the file indicated by path. Note that truncation permissions may also be checked based on already opened files, using the *security_file_truncate()* hook.

Return

Returns 0 if permission is granted.

int **security_path_chmod**(const struct *path* *path, umode_t mode)
    Check if changing the file's mode is allowed

Parameters

const struct path *path
    file

umode_t mode
    new mode

Description

Check for permission to change a mode of the file **path**. The new mode is specified in **mode** which is a bitmask of constants from <include/uapi/linux/stat.h>.

Return

Returns 0 if permission is granted.

int **security_path_chown**(const struct *path* *path, kuid_t uid, kgid_t gid)
    Check if changing the file's owner/group is allowed

Parameters

const struct path *path
    file

kuid_t uid
    file owner

kgid_t gid
    file group

Description

Check for permission to change owner/group of a file or directory.

Return

Returns 0 if permission is granted.

int **security_path_chroot**(const struct *path* *path)
    Check if changing the root directory is allowed

Parameters

const struct path *path
    directory

Description

Check for permission to change root directory.

Return

Returns 0 if permission is granted.

int **security_inode_link**(struct dentry *old_dentry, struct inode *dir, struct dentry *new_dentry)
    Check if creating a hard link is allowed

Parameters

struct dentry *old_dentry
    existing file

struct inode *dir
new parent directory

struct dentry *new_dentry
new link

Description

Check permission before creating a new hard link to a file.

Return

Returns 0 if permission is granted.

int **security_inode_unlink**(struct inode *dir, struct *dentry* *dentry)
Check if removing a hard link is allowed

Parameters

struct inode *dir
parent directory

struct dentry *dentry
file

Description

Check the permission to remove a hard link to a file.

Return

Returns 0 if permission is granted.

int **security_inode_symlink**(struct inode *dir, struct *dentry* *dentry, const char *old_name)
Check if creating a symbolic link is allowed

Parameters

struct inode *dir
parent directory

struct dentry *dentry
symbolic link

const char *old_name
existing filename

Description

Check the permission to create a symbolic link to a file.

Return

Returns 0 if permission is granted.

int **security_inode_rmdir**(struct inode *dir, struct *dentry* *dentry)
Check if removing a directory is allowed

Parameters

struct inode *dir
parent directory

struct dentry *dentry
directory to be removed

Description

Check the permission to remove a directory.

Return

Returns 0 if permission is granted.

int **security_inode_mknod**(struct inode *dir, struct *dentry* *dentry, umode_t mode, dev_t dev)
Check if creating a special file is allowed

Parameters

struct inode *dir
parent directory

struct dentry *dentry
new file

umode_t mode
new file mode

dev_t dev
device number

Description

Check permissions when creating a special file (or a socket or a fifo file created via the mknod system call). Note that if mknod operation is being done for a regular file, then the create hook will be called and not this hook.

Return

Returns 0 if permission is granted.

int **security_inode_rename**(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry, unsigned int flags)
Check if renaming a file is allowed

Parameters

struct inode *old_dir
parent directory of the old file

struct dentry *old_dentry
the old file

struct inode *new_dir
parent directory of the new file

struct dentry *new_dentry
the new file

unsigned int flags
flags

Description

Check for permission to rename a file or directory.

Return

Returns 0 if permission is granted.

int **security_inode_readlink**(struct *dentry* *dentry)

Check if reading a symbolic link is allowed

Parameters

struct dentry *dentry
link

Description

Check the permission to read the symbolic link.

Return

Returns 0 if permission is granted.

int **security_inode_follow_link**(struct *dentry* *dentry, struct *inode* *inode, bool rcu)

Check if following a symbolic link is allowed

Parameters

struct dentry *dentry
link dentry

struct inode *inode
link inode

bool rcu
true if in RCU-walk mode

Description

Check permission to follow a symbolic link when looking up a pathname. If **rcu** is true, **inode** is not stable.

Return

Returns 0 if permission is granted.

int **security_inode_permission**(struct *inode* *inode, int mask)

Check if accessing an inode is allowed

Parameters

struct inode *inode
inode

int mask
access mask

Description

Check permission before accessing an inode. This hook is called by the existing Linux permission function, so a security module can use it to provide additional checking for existing Linux permission checks. Notice that this hook is called when a file is opened (as well as many other operations), whereas the file_security_ops permission hook is called when the actual read/write operations are performed.

Return

Returns 0 if permission is granted.

int **security_inode_getattr**(const struct *path* *path)

Check if getting file attributes is allowed

Parameters

const struct path *path
file

Description

Check permission before obtaining file attributes.

Return

Returns 0 if permission is granted.

int **security_inode_setxattr**(struct mnt_idmap *idmap, struct *dentry* *dentry, const char *name, const void *value, size_t size, int flags)

Check if setting file xattrs is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

const char *name
xattr name

const void *value
xattr value

size_t size
size of xattr value

int flags
flags

Description

Check permission before setting the extended attributes.

Return

Returns 0 if permission is granted.

int **security_inode_set_acl**(struct mnt_idmap *idmap, struct *dentry* *dentry, const char *acl_name, struct posix_acl *kacl)

Check if setting posix acls is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

const char *acl_name
acl name

struct posix_acl *kacl
acl struct

Description

Check permission before setting posix acls, the posix acls in **kacl** are identified by **acl_name**.

Return

Returns 0 if permission is granted.

int **security_inode_get_acl**(struct mnt_idmap *idmap, struct *dentry* *dentry, const char *acl_name)

Check if reading posix acls is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

const char *acl_name
acl name

Description

Check permission before getting osix acls, the posix acls are identified by **acl_name**.

Return

Returns 0 if permission is granted.

int **security_inode_remove_acl**(struct mnt_idmap *idmap, struct *dentry* *dentry, const char *acl_name)

Check if removing a posix acl is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

const char *acl_name
acl name

Description

Check permission before removing posix acls, the posix acls are identified by **acl_name**.

Return

Returns 0 if permission is granted.

void **security_inode_post_setxattr**(struct *dentry* *dentry, const char *name, const void *value, size_t size, int flags)

Update the inode after a setxattr operation

Parameters

struct dentry *dentry
file

const char *name
xattr name

const void *value
xattr value

size_t size
xattr value size

int flags
flags

Description

Update inode security field after successful setxattr operation.

int **security_inode_getxattr**(struct *dentry* *dentry, const char *name)

Check if xattr access is allowed

Parameters

struct dentry *dentry
file

const char *name
xattr name

Description

Check permission before obtaining the extended attributes identified by **name** for **dentry**.

Return

Returns 0 if permission is granted.

int **security_inode_listxattr**(struct *dentry* *dentry)

Check if listing xattrs is allowed

Parameters

struct dentry *dentry
file

Description

Check permission before obtaining the list of extended attribute names for **dentry**.

Return

Returns 0 if permission is granted.

int **security_inode_removexattr**(struct mnt_idmap *idmap, struct *dentry* *dentry, const char *name)

Check if removing an xattr is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

const char *name
xattr name

Description

Check permission before removing the extended attribute identified by **name** for **dentry**.

Return

Returns 0 if permission is granted.

int **security_inode_need_killpriv**(struct *dentry* *dentry)

Check if *security_inode_killpriv()* required

Parameters

struct dentry *dentry
associated dentry

Description

Called when an inode has been changed to determine if *security_inode_killpriv()* should be called.

Return

Return <0 on error to abort the inode change operation, return 0 if
security_inode_killpriv() does not need to be called, return >0 if *security_inode_killpriv()* does need to be called.

int **security_inode_killpriv**(struct mnt_idmap *idmap, struct *dentry* *dentry)

The setuid bit is removed, update LSM state

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
associated dentry

Description

The **dentry**'s setuid bit is being removed. Remove similar security labels. Called with the dentry->d_inode->i_mutex held.

Return

Return 0 on success. If error is returned, then the operation
causing setuid bit removal is failed.

int **security_inode_getsecurity**(struct mnt_idmap *idmap, struct *inode* *inode, const char *name, void **buffer, bool alloc)

Get the xattr security label of an inode

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct inode *inode
inode

const char *name
xattr name

void **buffer
security label buffer

bool alloc
allocation flag

Description

Retrieve a copy of the extended attribute representation of the security label associated with **name** for **inode** via **buffer**. Note that **name** is the remainder of the attribute name after the security prefix has been removed. **alloc** is used to specify if the call should return a value via the buffer or just the value length.

Return

Returns size of buffer on success.

int **security_inode_setsecurity**(struct *inode* *inode, const char *name, const void *value, size_t size, int flags)

Set the xattr security label of an inode

Parameters

struct inode *inode
inode

const char *name
xattr name

const void *value
security label

size_t size
length of security label

int flags
flags

Description

Set the security label associated with **name** for **inode** from the extended attribute value **value**. **size** indicates the size of the **value** in bytes. **flags** may be XATTR_CREATE, XATTR_REPLACE, or 0. Note that **name** is the remainder of the attribute name after the security. prefix has been removed.

Return

Returns 0 on success.

void **security_inode_getsecid**(struct *inode* *inode, u32 *secid)

Get an inode's secid

Parameters

struct inode *inode
inode

u32 *secid
secid to return

Description

Get the secid associated with the node. In case of failure, **secid** will be set to zero.

int **security_kernfs_init_security**(struct kernfs_node *kn_dir, struct kernfs_node *kn)

Init LSM context for a kernfs node

Parameters

struct kernfs_node *kn_dir
parent kernfs node

struct kernfs_node *kn
the kernfs node to initialize

Description

Initialize the security context of a newly created kernfs node based on its own and its parent's attributes.

Return

Returns 0 if permission is granted.

int **security_file_permission**(struct *file* *file, int mask)

Check file permissions

Parameters

struct file *file
file

int mask
requested permissions

Description

Check file permissions before accessing an open file. This hook is called by various operations that read or write files. A security module can use this hook to perform additional checking on these operations, e.g. to revalidate permissions on use to support privilege bracketing or policy changes. Notice that this hook is used when the actual read/write operations are performed, whereas the inode_security_ops hook is called when a file is opened (as well as many other operations). Although this hook can be used to revalidate permissions for various system call operations that read or write files, it does not address the revalidation of permissions for memory-mapped files. Security modules must handle this separately if they need such revalidation.

Return

Returns 0 if permission is granted.

int **security_file_alloc**(struct *file* *file)

Allocate and init a file's LSM blob

Parameters

struct file *file
the file

Description

Allocate and attach a security structure to the file->f_security field. The security field is initialized to NULL when the structure is first created.

Return

Return 0 if the hook is successful and permission is granted.

void **security_file_free**(struct *file* *file)

Free a file's LSM blob

Parameters

struct file *file
the file

Description

Deallocate and free any security structures stored in file->f_security.

int **security_mmap_file**(struct *file* *file, unsigned long prot, unsigned long flags)

Check if mmap'ing a file is allowed

Parameters

struct file *file
file

unsigned long prot
protection applied by the kernel

unsigned long flags
flags

Description

Check permissions for a mmap operation. The **file** may be NULL, e.g. if mapping anonymous memory.

Return

Returns 0 if permission is granted.

int **security_mmap_addr**(unsigned long addr)

Check if mmap'ing an address is allowed

Parameters

unsigned long addr
address

Description

Check permissions for a mmap operation at **addr**.

Return

Returns 0 if permission is granted.

int **security_file_mprotect**(struct vm_area_struct *vma, unsigned long reqprot, unsigned long prot)

Check if changing memory protections is allowed

Parameters

struct vm_area_struct *vma
memory region

unsigned long reqprot
application requested protection

unsigned long prot
protection applied by the kernel

Description

Check permissions before changing memory access permissions.

Return

Returns 0 if permission is granted.

int **security_file_lock**(struct *file* *file, unsigned int cmd)

Check if a file lock is allowed

Parameters

struct file *file
file

unsigned int cmd
lock operation (e.g. F_RDLCK, F_WRLCK)

Description

Check permission before performing file locking operations. Note the hook mediates both flock and fcntl style locks.

Return

Returns 0 if permission is granted.

int **security_file_fcntl**(struct *file* *file, unsigned int cmd, unsigned long arg)

Check if fcntl() op is allowed

Parameters

struct file *file
file

unsigned int cmd
fcntl command

unsigned long arg
command argument

Description

Check permission before allowing the file operation specified by **cmd** from being performed on the file **file**. Note that **arg** sometimes represents a user space pointer; in other cases, it may be a simple integer value. When **arg** represents a user space pointer, it should never be used by the security module.

Return

Returns 0 if permission is granted.

void **security_file_set_fowner**(struct *file* *file)
Set the file owner info in the LSM blob

Parameters

struct file *file
the file

Description

Save owner security information (typically from current->security) in file->f_security for later use by the send_sigiotask hook.

Return

Returns 0 on success.

int **security_file_send_sigiotask**(struct task_struct *tsk, struct fown_struct *fown, int sig)
Check if sending SIGIO/SIGURG is allowed

Parameters

struct task_struct *tsk
target task

struct fown_struct *fown
signal sender

int sig
signal to be sent, SIGIO is sent if 0

Description

Check permission for the file owner **fown** to send SIGIO or SIGURG to the process **tsk**. Note that this hook is sometimes called from interrupt. Note that the fown_struct, **fown**, is never outside the context of a struct file, so the file structure (and associated security information) can always be obtained: container_of(fown, struct file, f_owner).

Return

Returns 0 if permission is granted.

int **security_file_receive**(struct *file* *file)
Check is receiving a file via IPC is allowed

Parameters

struct file *file
file being received

Description

This hook allows security modules to control the ability of a process to receive an open file descriptor via socket IPC.

Return

Returns 0 if permission is granted.

int **security_file_open**(struct *file* *file)
Save open() time state for late use by the LSM

Parameters

`struct file *file`

Description

Save open-time permission checking state for later use upon file_permission, and recheck access if anything has changed since inode_permission.

Return

Returns 0 if permission is granted.

int **security_file_truncate**(struct *file* *file)
Check if truncating a file is allowed

Parameters

struct file *file
file

Description

Check permission before truncating a file, i.e. using ftruncate. Note that truncation permission may also be checked based on the path, using the **path_truncate** hook.

Return

Returns 0 if permission is granted.

int **security_task_alloc**(struct task_struct *task, unsigned long clone_flags)
Allocate a task's LSM blob

Parameters

struct task_struct *task
the task

unsigned long clone_flags
flags indicating what is being shared

Description

Handle allocation of task-related resources.

Return

Returns a zero on success, negative values on failure.

void **security_task_free**(struct task_struct *task)
Free a task's LSM blob and related resources

Parameters

struct task_struct *task
task

Description

Handle release of task-related resources. Note that this can be called from interrupt context.

int **security_cred_alloc_blank**(struct *cred* *cred, gfp_t gfp)
Allocate the min memory to allow cred_transfer

Parameters

struct cred *cred
credentials

gfp_t gfp
gfp flags

Description

Only allocate sufficient memory and attach to **cred** such that cred_transfer() will not get ENOMEM.

Return

Returns 0 on success, negative values on failure.

void **security_cred_free**(struct *cred* *cred)
Free the cred's LSM blob and associated resources

Parameters

struct cred *cred
credentials

Description

Deallocate and clear the cred->security field in a set of credentials.

int **security_prepare_creds**(struct cred *new, const struct cred *old, gfp_t gfp)
Prepare a new set of credentials

Parameters

struct cred *new
new credentials

const struct cred *old
original credentials

gfp_t gfp
gfp flags

Description

Prepare a new set of credentials by copying the data from the old set.

Return

Returns 0 on success, negative values on failure.

void **security_transfer_creds**(struct cred *new, const struct cred *old)
Transfer creds

Parameters

struct cred *new
target credentials

const struct cred *old
original credentials

Description

Transfer data from original creds to new creds.

int **security_kernel_act_as**(struct cred *new, u32 secid)
Set the kernel credentials to act as secid

Parameters

struct cred *new
credentials

u32 secid
secid

Description

Set the credentials for a kernel service to act as (subjective context). The current task must be the one that nominated **secid**.

Return

Returns 0 if successful.

int **security_kernel_create_files_as**(struct cred *new, struct *inode* *inode)
Set file creation context using an inode

Parameters

struct cred *new
target credentials

struct inode *inode
reference inode

Description

Set the file creation context in a set of credentials to be the same as the objective context of the specified inode. The current task must be the one that nominated **inode**.

Return

Returns 0 if successful.

int **security_kernel_module_request**(char *kmod_name)
Check is loading a module is allowed

Parameters

char *kmod_name
module name

Description

Ability to trigger the kernel to automatically upcall to userspace for userspace to load a kernel module with the given name.

Return

Returns 0 if successful.

int **security_task_fix_setuid**(struct cred *new, const struct cred *old, int flags)
Update LSM with new user id attributes

Parameters

struct cred *new
updated credentials

const struct cred *old
credentials being replaced

int flags
LSM_SETID_* flag values

Description

Update the module's state after setting one or more of the user identity attributes of the current process. The **flags** parameter indicates which of the set*uid system calls invoked this hook. If **new** is the set of credentials that will be installed. Modifications should be made to this rather than to **current->cred**.

Return

Returns 0 on success.

int **security_task_fix_setgid**(struct cred *new, const struct cred *old, int flags)
Update LSM with new group id attributes

Parameters

struct cred *new
updated credentials

const struct cred *old
credentials being replaced

int flags
LSM_SETID_* flag value

Description

Update the module's state after setting one or more of the group identity attributes of the current process. The **flags** parameter indicates which of the set*gid system calls invoked this hook. **new** is the set of credentials that will be installed. Modifications should be made to this rather than to **current->cred**.

Return

Returns 0 on success.

int **security_task_fix_setgroups**(struct cred *new, const struct cred *old)
Update LSM with new supplementary groups

Parameters

struct cred *new
updated credentials

const struct cred *old
credentials being replaced

Description

Update the module's state after setting the supplementary group identity attributes of the current process. **new** is the set of credentials that will be installed. Modifications should be made to this rather than to **current->cred**.

Return

Returns 0 on success.

int **security_task_setpgid**(struct task_struct *p, pid_t pgid)
Check if setting the pgid is allowed

Parameters

struct task_struct *p
task being modified

pid_t pgid
new pgid

Description

Check permission before setting the process group identifier of the process **p** to **pgid**.

Return

Returns 0 if permission is granted.

int **security_task_getpgid**(struct task_struct *p)
Check if getting the pgid is allowed

Parameters

struct task_struct *p
task

Description

Check permission before getting the process group identifier of the process **p**.

Return

Returns 0 if permission is granted.

int **security_task_getsid**(struct task_struct *p)
Check if getting the session id is allowed

Parameters

struct task_struct *p
task

Description

Check permission before getting the session identifier of the process **p**.

Return

Returns 0 if permission is granted.

int **security_task_setnice**(struct task_struct *p, int nice)
Check if setting a task's nice value is allowed

Parameters

struct task_struct *p
target task

int nice
nice value

Description

Check permission before setting the nice value of **p** to **nice**.

Return

Returns 0 if permission is granted.

int **security_task_setioprio**(struct task_struct *p, int ioprio)
Check if setting a task's ioprio is allowed

Parameters

struct task_struct *p
target task

int ioprio
ioprio value

Description

Check permission before setting the ioprio value of **p** to **ioprio**.

Return

Returns 0 if permission is granted.

int **security_task_getioprio**(struct task_struct *p)
Check if getting a task's ioprio is allowed

Parameters

struct task_struct *p
task

Description

Check permission before getting the ioprio value of **p**.

Return

Returns 0 if permission is granted.

int **security_task_prlimit**(const struct *cred* *cred, const struct *cred* *tcred, unsigned int flags)

Check if get/setting resources limits is allowed

Parameters

const struct cred *cred
current task credentials

const struct cred *tcred
target task credentials

unsigned int flags
LSM_PRLIMIT_* flag bits indicating a get/set/both

Description

Check permission before getting and/or setting the resource limits of another task.

Return

Returns 0 if permission is granted.

int **security_task_setrlimit**(struct task_struct *p, unsigned int resource, struct rlimit *new_rlim)

Check if setting a new rlimit value is allowed

Parameters

struct task_struct *p
target task's group leader

unsigned int resource
resource whose limit is being set

struct rlimit *new_rlim
new resource limit

Description

Check permission before setting the resource limits of process **p** for **resource** to **new_rlim**. The old resource limit values can be examined by dereferencing (p->signal->rlim + resource).

Return

Returns 0 if permission is granted.

int **security_task_setscheduler**(struct task_struct *p)

Check if setting sched policy/param is allowed

Parameters

struct task_struct *p
target task

Description

Check permission before setting scheduling policy and/or parameters of process **p**.

Return

Returns 0 if permission is granted.

int **security_task_getscheduler**(struct task_struct *p)
Check if getting scheduling info is allowed

Parameters

struct task_struct *p
target task

Description

Check permission before obtaining scheduling information for process **p**.

Return

Returns 0 if permission is granted.

int **security_task_movememory**(struct task_struct *p)
Check if moving memory is allowed

Parameters

struct task_struct *p
task

Description

Check permission before moving memory owned by process **p**.

Return

Returns 0 if permission is granted.

int **security_task_kill**(struct task_struct *p, struct kernel_siginfo *info, int sig, const struct *cred* *cred)
Check if sending a signal is allowed

Parameters

struct task_struct *p
target process

struct kernel_siginfo *info
signal information

int sig
signal value

const struct cred *cred
credentials of the signal sender, NULL if **current**

Description

Check permission before sending signal **sig** to **p**. **info** can be NULL, the constant 1, or a pointer to a kernel_siginfo structure. If **info** is 1 or SI_FROMKERNEL(info) is true, then the signal should be viewed as coming from the kernel and should typically be permitted. SIGIO signals are handled separately by the send_sigiotask hook in file_security_ops.

Return

Returns 0 if permission is granted.

int **security_task_prctl**(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5)

Check if a prctl op is allowed

Parameters

int option
operation

unsigned long arg2
argument

unsigned long arg3
argument

unsigned long arg4
argument

unsigned long arg5
argument

Description

Check permission before performing a process control operation on the current process.

Return

Return -ENOSYS if no-one wanted to handle this op, any other value
to cause prctl() to return immediately with that value.

void **security_task_to_inode**(struct task_struct *p, struct *inode* *inode)

Set the security attributes of a task's inode

Parameters

struct task_struct *p
task

struct inode *inode
inode

Description

Set the security attributes for an inode based on an associated task's security attributes, e.g. for /proc/pid inodes.

int **security_create_user_ns**(const struct *cred* *cred)

Check if creating a new userns is allowed

Parameters

const struct cred *cred
prepared creds

Description

Check permission prior to creating a new user namespace.

Return

Returns 0 if successful, otherwise < 0 error code.

int **security_ipc_permission**(struct kern_ipc_perm *ipcp, short flag)
Check if sysv ipc access is allowed

Parameters

struct kern_ipc_perm *ipcp
ipc permission structure

short flag
requested permissions

Description

Check permissions for access to IPC.

Return

Returns 0 if permission is granted.

void **security_ipc_getsecid**(struct kern_ipc_perm *ipcp, u32 *secid)
Get the sysv ipc object's secid

Parameters

struct kern_ipc_perm *ipcp
ipc permission structure

u32 *secid
secid pointer

Description

Get the secid associated with the ipc object. In case of failure, **secid** will be set to zero.

int **security_msg_msg_alloc**(struct msg_msg *msg)
Allocate a sysv ipc message LSM blob

Parameters

struct msg_msg *msg
message structure

Description

Allocate and attach a security structure to the msg->security field. The security field is initialized to NULL when the structure is first created.

Return

Return 0 if operation was successful and permission is granted.

void **security_msg_msg_free**(struct msg_msg *msg)
Free a sysv ipc message LSM blob

Parameters

struct msg_msg *msg
message structure

Description

Deallocate the security structure for this message.

int **security_msg_queue_alloc**(struct kern_ipc_perm *msq)
Allocate a sysv ipc msg queue LSM blob

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

Description

Allocate and attach a security structure to **msg**. The security field is initialized to NULL when the structure is first created.

Return

Returns 0 if operation was successful and permission is granted.

void **security_msg_queue_free**(struct kern_ipc_perm *msq)
Free a sysv ipc msg queue LSM blob

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

Description

Deallocate security field **perm->security** for the message queue.

int **security_msg_queue_associate**(struct kern_ipc_perm *msq, int msqflg)
Check if a msg queue operation is allowed

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

int msqflg
operation flags

Description

Check permission when a message queue is requested through the msgget system call. This hook is only called when returning the message queue identifier for an existing message queue, not when a new message queue is created.

Return

Return 0 if permission is granted.

int **security_msg_queue_msgctl**(struct kern_ipc_perm *msq, int cmd)
Check if a msg queue operation is allowed

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

int cmd
operation

Description

Check permission when a message control operation specified by **cmd** is to be performed on the message queue with permissions.

Return

Returns 0 if permission is granted.

int **security_msg_queue_msgsnd**(struct kern_ipc_perm *msq, struct msg_msg *msg, int msqflg)

Check if sending a sysv ipc message is allowed

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

struct msg_msg *msg
message

int msqflg
operation flags

Description

Check permission before a message, **msg**, is enqueued on the message queue with permissions specified in **msq**.

Return

Returns 0 if permission is granted.

int **security_msg_queue_msgrcv**(struct kern_ipc_perm *msq, struct msg_msg *msg, struct task_struct *target, long type, int mode)

Check if receiving a sysv ipc msg is allowed

Parameters

struct kern_ipc_perm *msq
sysv ipc permission structure

struct msg_msg *msg
message

struct task_struct *target
target task

long type
type of message requested

int mode
operation flags

Description

Check permission before a message, **msg**, is removed from the message queue. The **target** task structure contains a pointer to the process that will be receiving the message (not equal to the current process when inline receives are being performed).

Return

Returns 0 if permission is granted.

int **security_shm_alloc**(struct kern_ipc_perm *shp)

Allocate a sysv shm LSM blob

Parameters

struct kern_ipc_perm *shp
sysv ipc permission structure

Description

Allocate and attach a security structure to the **shp** security field. The security field is initialized to NULL when the structure is first created.

Return

Returns 0 if operation was successful and permission is granted.

void **security_shm_free**(struct kern_ipc_perm *shp)

Free a sysv shm LSM blob

Parameters

struct kern_ipc_perm *shp
sysv ipc permission structure

Description

Deallocate the security structure **perm->security** for the memory segment.

int **security_shm_associate**(struct kern_ipc_perm *shp, int shmflg)

Check if a sysv shm operation is allowed

Parameters

struct kern_ipc_perm *shp
sysv ipc permission structure

int shmflg
operation flags

Description

Check permission when a shared memory region is requested through the shmget system call. This hook is only called when returning the shared memory region identifier for an existing region, not when a new shared memory region is created.

Return

Returns 0 if permission is granted.

int **security_shm_shmctl**(struct kern_ipc_perm *shp, int cmd)

Check if a sysv shm operation is allowed

Parameters

struct kern_ipc_perm *shp
sysv ipc permission structure

int cmd
operation

Description

Check permission when a shared memory control operation specified by **cmd** is to be performed on the shared memory region with permissions in **shp**.

Return

Return 0 if permission is granted.

int **security_shm_shmat**(struct kern_ipc_perm *shp, char __user *shmaddr, int shmflg)

Check if a sysv shm attach operation is allowed

Parameters

struct kern_ipc_perm *shp
sysv ipc permission structure

char __user *shmaddr
address of memory region to attach

int shmflg
operation flags

Description

Check permissions prior to allowing the shmat system call to attach the shared memory segment with permissions **shp** to the data segment of the calling process. The attaching address is specified by **shmaddr**.

Return

Returns 0 if permission is granted.

int **security_sem_alloc**(struct kern_ipc_perm *sma)

Allocate a sysv semaphore LSM blob

Parameters

struct kern_ipc_perm *sma
sysv ipc permission structure

Description

Allocate and attach a security structure to the **sma** security field. The security field is initialized to NULL when the structure is first created.

Return

Returns 0 if operation was successful and permission is granted.

void **security_sem_free**(struct kern_ipc_perm *sma)

Free a sysv semaphore LSM blob

Parameters

struct kern_ipc_perm *sma
sysv ipc permission structure

Description

Deallocate security structure **sma->security** for the semaphore.

int **security_sem_associate**(struct kern_ipc_perm *sma, int semflg)

Check if a sysv semaphore operation is allowed

Parameters

struct kern_ipc_perm *sma
sysv ipc permission structure

int semflg
operation flags

Description

Check permission when a semaphore is requested through the semget system call. This hook is only called when returning the semaphore identifier for an existing semaphore, not when a new one must be created.

Return

Returns 0 if permission is granted.

int **security_sem_semctl**(struct kern_ipc_perm *sma, int cmd)

Check if a sysv semaphore operation is allowed

Parameters

struct kern_ipc_perm *sma
sysv ipc permission structure

int cmd
operation

Description

Check permission when a semaphore operation specified by **cmd** is to be performed on the semaphore.

Return

Returns 0 if permission is granted.

int **security_sem_semop**(struct kern_ipc_perm *sma, struct sembuf *sops, unsigned nsops, int alter)

Check if a sysv semaphore operation is allowed

Parameters

struct kern_ipc_perm *sma
sysv ipc permission structure

struct sembuf *sops
operations to perform

unsigned nsops
number of operations

int alter
flag indicating changes will be made

Description

Check permissions before performing operations on members of the semaphore set. If the **alter** flag is nonzero, the semaphore set may be modified.

Return

Returns 0 if permission is granted.

int **security_getprocattr**(struct task_struct *p, const char *lsm, const char *name, char **value)

Read an attribute for a task

Parameters

struct task_struct *p
the task

const char *lsm
LSM name

const char *name
attribute name

char **value
attribute value

Description

Read attribute **name** for task **p** and store it into **value** if allowed.

Return

Returns the length of **value** on success, a negative value otherwise.

int **security_setprocattr**(const char *lsm, const char *name, void *value, size_t size)

Set an attribute for a task

Parameters

const char *lsm
LSM name

const char *name
attribute name

void *value
attribute value

size_t size
attribute value size

Description

Write (set) the current task's attribute **name** to **value**, size **size** if allowed.

Return

Returns bytes written on success, a negative value otherwise.

int **security_netlink_send**(struct sock *sk, struct sk_buff *skb)

Save info and check if netlink sending is allowed

Parameters

struct sock *sk
: sending socket

struct sk_buff *skb
: netlink message

Description

Save security information for a netlink message so that permission checking can be performed when the message is processed. The security information can be saved using the eff_cap field of the netlink_skb_parms structure. Also may be used to provide fine grained control over message transmission.

Return

Returns 0 if the information was successfully saved and message is
: allowed to be transmitted.

int **security_post_notification**(const struct *cred* *w_cred, const struct *cred* *cred, struct watch_notification *n)
: Check if a watch notification can be posted

Parameters

const struct cred *w_cred
: credentials of the task that set the watch

const struct cred *cred
: credentials of the task which triggered the watch

struct watch_notification *n
: the notification

Description

Check to see if a watch notification can be posted to a particular queue.

Return

Returns 0 if permission is granted.

int **security_watch_key**(struct *key* *key)
: Check if a task is allowed to watch for key events

Parameters

struct key *key
: the key to watch

Description

Check to see if a process is allowed to watch for event notifications from a key or keyring.

Return

Returns 0 if permission is granted.

int **security_socket_create**(int family, int type, int protocol, int kern)
: Check if creating a new socket is allowed

Parameters

int family
protocol family

int type
communications type

int protocol
requested protocol

int kern
set to 1 if a kernel socket is requested

Description

Check permissions prior to creating a new socket.

Return

Returns 0 if permission is granted.

int **security_socket_post_create**(struct socket *sock, int family, int type, int protocol, int kern)

Initialize a newly created socket

Parameters

struct socket *sock
socket

int family
protocol family

int type
communications type

int protocol
requested protocol

int kern
set to 1 if a kernel socket is requested

Description

This hook allows a module to update or allocate a per-socket security structure. Note that the security field was not added directly to the socket structure, but rather, the socket security information is stored in the associated inode. Typically, the inode alloc_security hook will allocate and attach security information to SOCK_INODE(sock)->i_security. This hook may be used to update the SOCK_INODE(sock)->i_security field with additional information that wasn't available when the inode was allocated.

Return

Returns 0 if permission is granted.

int **security_socket_bind**(struct socket *sock, struct sockaddr *address, int addrlen)

Check if a socket bind operation is allowed

Parameters

struct socket *sock
socket

struct sockaddr *address
requested bind address

int addrlen
length of address

Description

Check permission before socket protocol layer bind operation is performed and the socket **sock** is bound to the address specified in the **address** parameter.

Return

Returns 0 if permission is granted.

int **security_socket_connect**(struct socket *sock, struct sockaddr *address, int addrlen)
Check if a socket connect operation is allowed

Parameters

struct socket *sock
socket

struct sockaddr *address
address of remote connection point

int addrlen
length of address

Description

Check permission before socket protocol layer connect operation attempts to connect socket **sock** to a remote address, **address**.

Return

Returns 0 if permission is granted.

int **security_socket_listen**(struct socket *sock, int backlog)
Check if a socket is allowed to listen

Parameters

struct socket *sock
socket

int backlog
connection queue size

Description

Check permission before socket protocol layer listen operation.

Return

Returns 0 if permission is granted.

int **security_socket_accept**(struct socket *sock, struct socket *newsock)
Check if a socket is allowed to accept connections

Parameters

struct socket *sock
listening socket

struct socket *newsock
newly creation connection socket

Description

Check permission before accepting a new connection. Note that the new socket, **newsock**, has been created and some information copied to it, but the accept operation has not actually been performed.

Return

Returns 0 if permission is granted.

int **security_socket_sendmsg**(struct socket *sock, struct msghdr *msg, int size)
Check is sending a message is allowed

Parameters

struct socket *sock
sending socket

struct msghdr *msg
message to send

int size
size of message

Description

Check permission before transmitting a message to another socket.

Return

Returns 0 if permission is granted.

int **security_socket_recvmsg**(struct socket *sock, struct msghdr *msg, int size, int flags)
Check if receiving a message is allowed

Parameters

struct socket *sock
receiving socket

struct msghdr *msg
message to receive

int size
size of message

int flags
operational flags

Description

Check permission before receiving a message from a socket.

Return

Returns 0 if permission is granted.

int **security_socket_getsockname**(struct socket *sock)
Check if reading the socket addr is allowed

Parameters

struct socket *sock
socket

Description

Check permission before reading the local address (name) of the socket object.

Return

Returns 0 if permission is granted.

int **security_socket_getpeername**(struct socket *sock)
Check if reading the peer's addr is allowed

Parameters

struct socket *sock
socket

Description

Check permission before the remote address (name) of a socket object.

Return

Returns 0 if permission is granted.

int **security_socket_getsockopt**(struct socket *sock, int level, int optname)
Check if reading a socket option is allowed

Parameters

struct socket *sock
socket

int level
option's protocol level

int optname
option name

Description

Check permissions before retrieving the options associated with socket **sock**.

Return

Returns 0 if permission is granted.

int **security_socket_setsockopt**(struct socket *sock, int level, int optname)
Check if setting a socket option is allowed

Parameters

struct socket *sock
socket

int level
option’s protocol level

int optname
option name

Description

Check permissions before setting the options associated with socket **sock**.

Return

Returns 0 if permission is granted.

int **security_socket_shutdown**(struct socket *sock, int how)
Checks if shutting down the socket is allowed

Parameters

struct socket *sock
socket

int how
flag indicating how sends and receives are handled

Description

Checks permission before all or part of a connection on the socket **sock** is shut down.

Return

Returns 0 if permission is granted.

int **security_socket_getpeersec_stream**(struct socket *sock, sockptr_t optval, sockptr_t optlen, unsigned int len)
Get the remote peer label

Parameters

struct socket *sock
socket

sockptr_t optval
destination buffer

sockptr_t optlen
size of peer label copied into the buffer

unsigned int len
maximum size of the destination buffer

Description

This hook allows the security module to provide peer socket security state for unix or connected tcp sockets to userspace via getsockopt SO_GETPEERSEC. For tcp sockets this can be meaningful if the socket is associated with an ipsec SA.

Return

Returns 0 if all is well, otherwise, typical getsockopt return
values.

int **security_sk_alloc**(struct sock *sk, int family, gfp_t priority)
Allocate and initialize a sock's LSM blob

Parameters

struct sock *sk
sock

int family
protocol family

gfp_t priority
gfp flags

Description

Allocate and attach a security structure to the sk->sk_security field, which is used to copy security attributes between local stream sockets.

Return

Returns 0 on success, error on failure.

void **security_sk_free**(struct sock *sk)
Free the sock's LSM blob

Parameters

struct sock *sk
sock

Description

Deallocate security structure.

void **security_inet_csk_clone**(struct sock *newsk, const struct request_sock *req)
Set new sock LSM state based on request_sock

Parameters

struct sock *newsk
new sock

const struct request_sock *req
connection request_sock

Description

Set that LSM state of **sock** using the LSM state from **req**.

int **security_mptcp_add_subflow**(struct sock *sk, struct sock *ssk)
Inherit the LSM label from the MPTCP socket

Parameters

struct sock *sk
the owning MPTCP socket

struct sock *ssk
the new subflow

Description

Update the labeling for the given MPTCP subflow, to match the one of the owning MPTCP socket. This hook has to be called after the socket creation and initialization via the *security_socket_create()* and *security_socket_post_create()* LSM hooks.

Return

Returns 0 on success or a negative error code on failure.

int **security_xfrm_policy_clone**(struct xfrm_sec_ctx *old_ctx, struct xfrm_sec_ctx **new_ctxp)

Clone xfrm policy LSM state

Parameters

struct xfrm_sec_ctx *old_ctx
xfrm security context

struct xfrm_sec_ctx **new_ctxp
target xfrm security context

Description

Allocate a security structure in new_ctxp that contains the information from the old_ctx structure.

Return

Return 0 if operation was successful.

int **security_xfrm_policy_delete**(struct xfrm_sec_ctx *ctx)

Check if deleting a xfrm policy is allowed

Parameters

struct xfrm_sec_ctx *ctx
xfrm security context

Description

Authorize deletion of a SPD entry.

Return

Returns 0 if permission is granted.

int **security_xfrm_state_alloc_acquire**(struct xfrm_state *x, struct xfrm_sec_ctx *polsec, u32 secid)

Allocate a xfrm state LSM blob

Parameters

struct xfrm_state *x
xfrm state being added to the SAD

struct xfrm_sec_ctx *polsec
associated policy's security context

u32 secid
secid from the flow

Description

Allocate a security structure to the x->security field; the security field is initialized to NULL when the xfrm_state is allocated. Set the context to correspond to secid.

Return

Returns 0 if operation was successful.

void **security_xfrm_state_free**(struct xfrm_state *x)

Free a xfrm state

Parameters

struct xfrm_state *x
xfrm state

Description

Deallocate x->security.

int **security_xfrm_policy_lookup**(struct xfrm_sec_ctx *ctx, u32 fl_secid)

Check if using a xfrm policy is allowed

Parameters

struct xfrm_sec_ctx *ctx
target xfrm security context

u32 fl_secid
flow secid used to authorize access

Description

Check permission when a flow selects a xfrm_policy for processing XFRMs on a packet. The hook is called when selecting either a per-socket policy or a generic xfrm policy.

Return

Return 0 if permission is granted, -ESRCH otherwise, or -errno on
other errors.

int **security_xfrm_state_pol_flow_match**(struct xfrm_state *x, struct xfrm_policy *xp, const struct flowi_common *flic)

Check for a xfrm match

Parameters

struct xfrm_state *x
xfrm state to match

struct xfrm_policy *xp
xfrm policy to check for a match

const struct flowi_common *flic
flow to check for a match.

Description

Check **xp** and **flic** for a match with **x**.

Return

Returns 1 if there is a match.

int **security_xfrm_decode_session**(struct sk_buff *skb, u32 *secid)
Determine the xfrm secid for a packet

Parameters

struct sk_buff *skb
xfrm packet

u32 *secid
secid

Description

Decode the packet in **skb** and return the security label in **secid**.

Return

Return 0 if all xfrms used have the same secid.

int **security_key_alloc**(struct *key* *key, const struct *cred* *cred, unsigned long flags)
Allocate and initialize a kernel key LSM blob

Parameters

struct key *key
key

const struct cred *cred
credentials

unsigned long flags
allocation flags

Description

Permit allocation of a key and assign security data. Note that key does not have a serial number assigned at this point.

Return

Return 0 if permission is granted, -ve error otherwise.

void **security_key_free**(struct *key* *key)
Free a kernel key LSM blob

Parameters

struct key *key
key

Description

Notification of destruction; free security data.

int **security_key_permission**(key_ref_t key_ref, const struct *cred* *cred, enum key_need_perm need_perm)
Check if a kernel key operation is allowed

Parameters

key_ref_t key_ref
key reference

const struct cred *cred
credentials of actor requesting access

enum key_need_perm need_perm
requested permissions

Description

See whether a specific operational right is granted to a process on a key.

Return

Return 0 if permission is granted, -ve error otherwise.

int **security_key_getsecurity**(struct *key* *key, char **buffer)
Get the key's security label

Parameters

struct key *key
key

char **buffer
security label buffer

Description

Get a textual representation of the security context attached to a key for the purposes of honouring KEYCTL_GETSECURITY. This function allocates the storage for the NUL-terminated string and the caller should free it.

Return

Returns the length of buffer (including terminating NUL) or -ve if
an error occurs. May also return 0 (and a NULL buffer pointer) if there is no security label assigned to the key.

int **security_audit_rule_init**(u32 field, u32 op, char *rulestr, void **lsmrule)
Allocate and init an LSM audit rule struct

Parameters

u32 field
audit action

u32 op
rule operator

char *rulestr
rule context

void **lsmrule
receive buffer for audit rule struct

Description

Allocate and initialize an LSM audit rule structure.

Return

Return 0 if lsmrule has been successfully set, -EINVAL in case of
an invalid rule.

int **security_audit_rule_known**(struct audit_krule *krule)
Check if an audit rule contains LSM fields

Parameters

struct audit_krule *krule
audit rule

Description

Specifies whether given **krule** contains any fields related to the current LSM.

Return

Returns 1 in case of relation found, 0 otherwise.

void **security_audit_rule_free**(void *lsmrule)
Free an LSM audit rule struct

Parameters

void *lsmrule
audit rule struct

Description

Deallocate the LSM audit rule structure previously allocated by audit_rule_init().

int **security_audit_rule_match**(u32 secid, u32 field, u32 op, void *lsmrule)
Check if a label matches an audit rule

Parameters

u32 secid
security label

u32 field
LSM audit field

u32 op
matching operator

void *lsmrule
audit rule

Description

Determine if given **secid** matches a rule previously approved by *security_audit_rule_known()*.

Return

Returns 1 if secid matches the rule, 0 if it does not, -ERRNO on
failure.

int **security_bpf**(int cmd, union bpf_attr *attr, unsigned int size)
Check if the bpf syscall operation is allowed

Parameters

int cmd
command

union bpf_attr *attr
bpf attribute

unsigned int size
size

Description

Do a initial check for all bpf syscalls after the attribute is copied into the kernel. The actual security module can implement their own rules to check the specific cmd they need.

Return

Returns 0 if permission is granted.

int **security_bpf_map**(struct bpf_map *map, fmode_t fmode)
Check if access to a bpf map is allowed

Parameters

struct bpf_map *map
bpf map

fmode_t fmode
mode

Description

Do a check when the kernel generates and returns a file descriptor for eBPF maps.

Return

Returns 0 if permission is granted.

int **security_bpf_prog**(struct bpf_prog *prog)
Check if access to a bpf program is allowed

Parameters

struct bpf_prog *prog
bpf program

Description

Do a check when the kernel generates and returns a file descriptor for eBPF programs.

Return

Returns 0 if permission is granted.

int **security_bpf_map_alloc**(struct bpf_map *map)
Allocate a bpf map LSM blob

Parameters

struct bpf_map *map
bpf map

Description

Initialize the security field inside bpf map.

Return

Returns 0 on success, error on failure.

int **security_bpf_prog_alloc**(struct bpf_prog_aux *aux)
Allocate a bpf program LSM blob

Parameters

struct bpf_prog_aux *aux
bpf program aux info struct

Description

Initialize the security field inside bpf program.

Return

Returns 0 on success, error on failure.

void **security_bpf_map_free**(struct bpf_map *map)
Free a bpf map's LSM blob

Parameters

struct bpf_map *map
bpf map

Description

Clean up the security information stored inside bpf map.

void **security_bpf_prog_free**(struct bpf_prog_aux *aux)
Free a bpf program's LSM blob

Parameters

struct bpf_prog_aux *aux
bpf program aux info struct

Description

Clean up the security information stored inside bpf prog.

int **security_perf_event_open**(struct perf_event_attr *attr, int type)
Check if a perf event open is allowed

Parameters

struct perf_event_attr *attr
perf event attribute

int type
type of event

Description

Check whether the **type** of perf_event_open syscall is allowed.

Return

Returns 0 if permission is granted.

int **security_perf_event_alloc**(struct perf_event *event)
Allocate a perf event LSM blob

Parameters

struct perf_event *event
perf event

Description

Allocate and save perf_event security info.

Return

Returns 0 on success, error on failure.

void **security_perf_event_free**(struct perf_event *event)
Free a perf event LSM blob

Parameters

struct perf_event *event
perf event

Description

Release (free) perf_event security info.

int **security_perf_event_read**(struct perf_event *event)
Check if reading a perf event label is allowed

Parameters

struct perf_event *event
perf event

Description

Read perf_event security info if allowed.

Return

Returns 0 if permission is granted.

int **security_perf_event_write**(struct perf_event *event)
Check if writing a perf event label is allowed

Parameters

struct perf_event *event
perf event

Description

Write perf_event security info if allowed.

Return

Returns 0 if permission is granted.

int **security_uring_override_creds**(const struct cred *new)
Check if overriding creds is allowed

Parameters

const struct cred *new
new credentials

Description

Check if the current task, executing an io_uring operation, is allowed to override it's credentials with **new**.

Return

Returns 0 if permission is granted.

int **security_uring_sqpoll**(void)
Check if IORING_SETUP_SQPOLL is allowed

Parameters

void
no arguments

Description

Check whether the current task is allowed to spawn a io_uring polling thread (IORING_SETUP_SQPOLL).

Return

Returns 0 if permission is granted.

int **security_uring_cmd**(struct io_uring_cmd *ioucmd)
Check if a io_uring passthrough command is allowed

Parameters

struct io_uring_cmd *ioucmd
command

Description

Check whether the file_operations uring_cmd is allowed to run.

Return

Returns 0 if permission is granted.

struct dentry ***securityfs_create_file**(const char *name, umode_t mode, struct dentry *parent, void *data, const struct file_operations *fops)
create a file in the securityfs filesystem

Parameters

const char *name
a pointer to a string containing the name of the file to create.

umode_t mode
the permission that the file should have

struct dentry *parent
a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the securityfs filesystem.

void *data
: a pointer to something that the caller will want to get to later on. The inode.i_private pointer will point to this value on the open() call.

const struct file_operations *fops
: a pointer to a struct file_operations that should be used for this file.

Description

This function creates a file in securityfs with the given **name**.

This function returns a pointer to a dentry if it succeeds. This pointer must be passed to the *securityfs_remove()* function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here). If an error occurs, the function will return the error value (via ERR_PTR).

If securityfs is not enabled in the kernel, the value `-ENODEV` is returned.

struct dentry ***securityfs_create_dir**(const char *name, struct dentry *parent)
: create a directory in the securityfs filesystem

Parameters

const char *name
: a pointer to a string containing the name of the directory to create.

struct dentry *parent
: a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the directory will be created in the root of the securityfs filesystem.

Description

This function creates a directory in securityfs with the given **name**.

This function returns a pointer to a dentry if it succeeds. This pointer must be passed to the *securityfs_remove()* function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here). If an error occurs, the function will return the error value (via ERR_PTR).

If securityfs is not enabled in the kernel, the value `-ENODEV` is returned.

struct dentry ***securityfs_create_symlink**(const char *name, struct dentry *parent, const char *target, const struct inode_operations *iops)
: create a symlink in the securityfs filesystem

Parameters

const char *name
: a pointer to a string containing the name of the symlink to create.

struct dentry *parent
: a pointer to the parent dentry for the symlink. This should be a directory dentry if set. If this parameter is `NULL`, then the directory will be created in the root of the securityfs filesystem.

const char *target
: a pointer to a string containing the name of the symlink's target. If this parameter is `NULL`, then the **iops** parameter needs to be setup to handle .readlink and .get_link inode_operations.

const struct inode_operations *iops
a pointer to the struct inode_operations to use for the symlink. If this parameter is NULL, then the default simple_symlink_inode operations will be used.

Description

This function creates a symlink in securityfs with the given **name**.

This function returns a pointer to a dentry if it succeeds. This pointer must be passed to the *securityfs_remove()* function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here). If an error occurs, the function will return the error value (via ERR_PTR).

If securityfs is not enabled in the kernel, the value -ENODEV is returned.

void **securityfs_remove**(struct *dentry* *dentry)
removes a file or directory from the securityfs filesystem

Parameters

struct dentry *dentry
a pointer to a the dentry of the file or directory to be removed.

Description

This function removes a file or directory in securityfs that was previously created with a call to another securityfs function (like *securityfs_create_file()* or variants thereof.)

This function is required to be called in order for the file to be removed. No automatic cleanup of files will happen when a module is removed; you are responsible here.

1.1.11 Audit Interfaces

struct audit_buffer ***audit_log_start**(struct audit_context *ctx, gfp_t gfp_mask, int type)
obtain an audit buffer

Parameters

struct audit_context *ctx
audit_context (may be NULL)

gfp_t gfp_mask
type of allocation

int type
audit message type

Description

Returns audit_buffer pointer on success or NULL on error.

Obtain an audit buffer. This routine does locking to obtain the audit buffer, but then no locking is required for calls to audit_log_*format. If the task (ctx) is a task that is currently in a syscall, then the syscall is marked as auditable and an audit record will be written at syscall exit. If there is no associated task, then task context (ctx) should be NULL.

void **audit_log_format**(struct audit_buffer *ab, const char *fmt, ...)
format a message into the audit buffer.

Parameters

struct audit_buffer *ab
audit_buffer

const char *fmt
format string

...
optional parameters matching **fmt** string

Description

All the work is done in audit_log_vformat.

void **audit_log_end**(struct audit_buffer *ab)
end one audit record

Parameters

struct audit_buffer *ab
the audit_buffer

Description

We can not do a netlink send inside an irq context because it blocks (last arg, flags, is not set to MSG_DONTWAIT), so the audit buffer is placed on a queue and a kthread is scheduled to remove them from the queue outside the irq context. May be called in any context.

void **audit_log**(struct audit_context *ctx, gfp_t gfp_mask, int type, const char *fmt, ...)
Log an audit record

Parameters

struct audit_context *ctx
audit context

gfp_t gfp_mask
type of allocation

int type
audit message type

const char *fmt
format string to use

...
variable parameters matching the format string

Description

This is a convenience function that calls audit_log_start, audit_log_vformat, and audit_log_end. It may be called in any context.

int **__audit_filter_op**(struct task_struct *tsk, struct audit_context *ctx, struct list_head *list, struct audit_names *name, unsigned long op)
common filter helper for operations (syscall/uring/etc)

Parameters

struct task_struct *tsk
associated task

struct audit_context *ctx
audit context

struct list_head *list
audit filter list

struct audit_names *name
audit_name (can be NULL)

unsigned long op
current syscall/uring_op

Description

Run the udit filters specified in **list** against **tsk** using **ctx**, **name**, and **op**, as necessary; the caller is responsible for ensuring that the call is made while the RCU read lock is held. The **name** parameter can be NULL, but all others must be specified. Returns 1/true if the filter finds a match, 0/false if none are found.

void **audit_filter_uring**(struct task_struct *tsk, struct audit_context *ctx)
apply filters to an io_uring operation

Parameters

struct task_struct *tsk
associated task

struct audit_context *ctx
audit context

void **audit_reset_context**(struct audit_context *ctx)
reset a audit_context structure

Parameters

struct audit_context *ctx
the audit_context to reset

Description

All fields in the audit_context will be reset to an initial state, all references held by fields will be dropped, and private memory will be released. When this function returns the audit_context will be suitable for reuse, so long as the passed context is not NULL or a dummy context.

int **audit_alloc**(struct task_struct *tsk)
allocate an audit context block for a task

Parameters

struct task_struct *tsk
task

Description

Filter on the task information and allocate a per-task audit context if necessary. Doing so turns on system call auditing for the specified task. This is called from copy_process, so no lock is needed.

void **audit_log_uring**(struct audit_context *ctx)
generate a AUDIT_URINGOP record

Parameters

struct audit_context *ctx
: the audit context

void **__audit_free**(struct task_struct *tsk)
: free a per-task audit context

Parameters

struct task_struct *tsk
: task whose audit context block to free

Description

Called from copy_process, do_exit, and the io_uring code

void **audit_return_fixup**(struct audit_context *ctx, int success, long code)
: fixup the return codes in the audit_context

Parameters

struct audit_context *ctx
: the audit_context

int success
: true/false value to indicate if the operation succeeded or not

long code
: operation return code

Description

We need to fixup the return code in the audit logs if the actual return codes are later going to be fixed by the arch specific signal handlers.

void **__audit_uring_entry**(u8 op)
: prepare the kernel task's audit context for io_uring

Parameters

u8 op
: the io_uring opcode

Description

This is similar to audit_syscall_entry() but is intended for use by io_uring operations. This function should only ever be called from audit_uring_entry() as we rely on the audit context checking present in that function.

void **__audit_uring_exit**(int success, long code)
: wrap up the kernel task's audit context after io_uring

Parameters

int success
: true/false value to indicate if the operation succeeded or not

long code
: operation return code

Description

This is similar to audit_syscall_exit() but is intended for use by io_uring operations. This function should only ever be called from audit_uring_exit() as we rely on the audit context checking present in that function.

void **__audit_syscall_entry**(int major, unsigned long a1, unsigned long a2, unsigned long a3, unsigned long a4)

fill in an audit record at syscall entry

Parameters

int major
major syscall type (function)

unsigned long a1
additional syscall register 1

unsigned long a2
additional syscall register 2

unsigned long a3
additional syscall register 3

unsigned long a4
additional syscall register 4

Description

Fill in audit context at syscall entry. This only happens if the audit context was created when the task was created and the state or filters demand the audit context be built. If the state from the per-task filter or from the per-syscall filter is AUDIT_STATE_RECORD, then the record will be written at syscall exit time (otherwise, it will only be written if another part of the kernel requests that it be written).

void **__audit_syscall_exit**(int success, long return_code)

deallocate audit context after a system call

Parameters

int success
success value of the syscall

long return_code
return value of the syscall

Description

Tear down after system call. If the audit context has been marked as auditable (either because of the AUDIT_STATE_RECORD state from filtering, or because some other part of the kernel wrote an audit message), then write out the syscall information. In call cases, free the names stored from getname().

struct filename ***__audit_reusename**(__user const char *uptr)

fill out filename with info from existing entry

Parameters

const __user char *uptr
userland ptr to pathname

Description

Search the audit_names list for the current audit context. If there is an existing entry with a matching "uptr" then return the filename associated with that audit_name. If not, return NULL.

void **__audit_getname**(struct filename *name)
add a name to the list

Parameters

struct filename *name
name to add

Description

Add a name to the list of audit names for this context. Called from fs/namei.c:getname().

void **__audit_inode**(struct filename *name, const struct *dentry* *dentry, unsigned int flags)
store the inode and device from a lookup

Parameters

struct filename *name
name being audited

const struct dentry *dentry
dentry being audited

unsigned int flags
attributes for this particular entry

int **auditsc_get_stamp**(struct audit_context *ctx, struct timespec64 *t, unsigned int *serial)
get local copies of audit_context values

Parameters

struct audit_context *ctx
audit_context for the task

struct timespec64 *t
timespec64 to store time recorded in the audit_context

unsigned int *serial
serial value that is recorded in the audit_context

Description

Also sets the context as auditable.

void **__audit_mq_open**(int oflag, umode_t mode, struct mq_attr *attr)
record audit data for a POSIX MQ open

Parameters

int oflag
open flag

umode_t mode
mode bits

struct mq_attr *attr
queue attributes

void **__audit_mq_sendrecv**(mqd_t mqdes, size_t msg_len, unsigned int msg_prio, const struct timespec64 *abs_timeout)

record audit data for a POSIX MQ timed send/receive

Parameters

mqd_t mqdes
MQ descriptor

size_t msg_len
Message length

unsigned int msg_prio
Message priority

const struct timespec64 *abs_timeout
Message timeout in absolute time

void **__audit_mq_notify**(mqd_t mqdes, const struct sigevent *notification)

record audit data for a POSIX MQ notify

Parameters

mqd_t mqdes
MQ descriptor

const struct sigevent *notification
Notification event

void **__audit_mq_getsetattr**(mqd_t mqdes, struct mq_attr *mqstat)

record audit data for a POSIX MQ get/set attribute

Parameters

mqd_t mqdes
MQ descriptor

struct mq_attr *mqstat
MQ flags

void **__audit_ipc_obj**(struct kern_ipc_perm *ipcp)

record audit data for ipc object

Parameters

struct kern_ipc_perm *ipcp
ipc permissions

void **__audit_ipc_set_perm**(unsigned long qbytes, uid_t uid, gid_t gid, umode_t mode)

record audit data for new ipc permissions

Parameters

unsigned long qbytes
msgq bytes

uid_t uid
msgq user id

gid_t gid
msgq group id

umode_t mode
msgq mode (permissions)

Description

Called only after audit_ipc_obj().

int **__audit_socketcall**(int nargs, unsigned long *args)
record audit data for sys_socketcall

Parameters

int nargs
number of args, which should not be more than AUDITSC_ARGS.

unsigned long *args
args array

void **__audit_fd_pair**(int fd1, int fd2)
record audit data for pipe and socketpair

Parameters

int fd1
the first file descriptor

int fd2
the second file descriptor

int **__audit_sockaddr**(int len, void *a)
record audit data for sys_bind, sys_connect, sys_sendto

Parameters

int len
data length in user space

void *a
data address in kernel space

Description

Returns 0 for success or NULL context or < 0 on error.

int **audit_signal_info_syscall**(struct task_struct *t)
record signal info for syscalls

Parameters

struct task_struct *t
task being signaled

Description

If the audit subsystem is being terminated, record the task (pid) and uid that is doing that.

int **__audit_log_bprm_fcaps**(struct linux_binprm *bprm, const struct cred *new, const struct cred *old)
store information about a loading bprm and relevant fcaps

Parameters

struct linux_binprm *bprm
pointer to the bprm being processed

const struct cred *new
the proposed new credentials

const struct cred *old
the old credentials

Description

Simply check if the proc already has the caps given by the file and if not store the priv escalation info for later auditing at the end of the syscall

-Eric

void **__audit_log_capset**(const struct cred *new, const struct cred *old)
store information about the arguments to the capset syscall

Parameters

const struct cred *new
the new credentials

const struct cred *old
the old (current) credentials

Description

Record the arguments userspace sent to sys_capset for later printing by the audit system if applicable

void **audit_core_dumps**(long signr)
record information about processes that end abnormally

Parameters

long signr
signal value

Description

If a process ends with a core dump, something fishy is going on and we should record the event for investigation.

void **audit_seccomp**(unsigned long syscall, long signr, int code)
record information about a seccomp action

Parameters

unsigned long syscall
syscall number

long signr
signal value

int code
the seccomp action

Description

Record the information associated with a seccomp action. Event filtering for seccomp actions that are not to be logged is done in seccomp_log(). Therefore, this function forces auditing independent of the audit_enabled and dummy context state because seccomp actions should be logged even when audit is not in use.

int **audit_rule_change**(int type, int seq, void *data, size_t datasz)

apply all rules to the specified message type

Parameters

int type
audit message type

int seq
netlink audit message sequence (serial) number

void *data
payload data

size_t datasz
size of payload data

int **audit_list_rules_send**(struct sk_buff *request_skb, int seq)

list the audit rules

Parameters

struct sk_buff *request_skb
skb of request we are replying to (used to target the reply)

int seq
netlink audit message sequence (serial) number

int **parent_len**(const char *path)

find the length of the parent portion of a pathname

Parameters

const char *path
pathname of which to determine length

int **audit_compare_dname_path**(const struct qstr *dname, const char *path, int parentlen)

compare given dentry name with last component in given path. Return of 0 indicates a match.

Parameters

const struct qstr *dname
dentry name that we're comparing

const char *path
full pathname that we're comparing

int parentlen
length of the parent if known. Passing in AUDIT_NAME_FULL here indicates that we must compute this value.

1.1.12 Accounting Framework

long **sys_acct**(const char __user *name)
enable/disable process accounting

Parameters

const char __user * name
file name for accounting records or NULL to shutdown accounting

Description

sys_acct() is the only system call needed to implement process accounting. It takes the name of the file where accounting records should be written. If the filename is NULL, accounting will be shutdown.

Return

0 for success or negative errno values for failure.

void **acct_collect**(long exitcode, int group_dead)
collect accounting information into pacct_struct

Parameters

long exitcode
task exit code

int group_dead
not 0, if this thread is the last one in the process.

void **acct_process**(void)
handles process accounting for an exiting task

Parameters

void
no arguments

1.1.13 Block Devices

void **bio_advance**(struct *bio* *bio, unsigned int nbytes)
increment/complete a bio by some number of bytes

Parameters

struct bio *bio
bio to advance

unsigned int nbytes
number of bytes to complete

Description

This updates bi_sector, bi_size and bi_idx; if the number of bytes to complete doesn't align with a bvec boundary, then bv_len and bv_offset will be updated on the last bvec as well.

bio will then represent the remaining, uncompleted portion of the io.

struct **folio_iter**

State for iterating all folios in a bio.

Definition:

```
struct folio_iter {
    struct folio *folio;
    size_t offset;
    size_t length;
};
```

Members

folio
The current folio we're iterating. NULL after the last folio.

offset
The byte offset within the current folio.

length
The number of bytes in this iteration (will not cross folio boundary).

bio_for_each_folio_all

`bio_for_each_folio_all (fi, bio)`

Iterate over each folio in a bio.

Parameters

fi
struct folio_iter which is updated for each folio.

bio
struct bio to iterate over.

struct *bio* ***bio_next_split**(struct *bio* *bio, int sectors, gfp_t gfp, struct bio_set *bs)

get next **sectors** from a bio, splitting if necessary

Parameters

struct bio *bio
bio to split

int sectors
number of sectors to split from the front of **bio**

gfp_t gfp
gfp mask

struct bio_set *bs
bio set to allocate from

Return

a bio representing the next **sectors** of **bio** - if the bio is smaller than **sectors**, returns the original bio unchanged.

void **blk_queue_flag_set**(unsigned int flag, struct request_queue *q)

atomically set a queue flag

Parameters

unsigned int flag
flag to be set

struct request_queue *q
request queue

void **blk_queue_flag_clear**(unsigned int flag, struct request_queue *q)
atomically clear a queue flag

Parameters

unsigned int flag
flag to be cleared

struct request_queue *q
request queue

bool **blk_queue_flag_test_and_set**(unsigned int flag, struct request_queue *q)
atomically test and set a queue flag

Parameters

unsigned int flag
flag to be set

struct request_queue *q
request queue

Description

Returns the previous value of **flag** - 0 if the flag was not set and 1 if the flag was already set.

const char ***blk_op_str**(enum req_op op)
Return string XXX in the REQ_OP_XXX.

Parameters

enum req_op op
REQ_OP_XXX.

Description

Centralize block layer function to convert REQ_OP_XXX into string format. Useful in the debugging and tracing bio or request. For invalid REQ_OP_XXX it returns string "UNKNOWN".

void **blk_sync_queue**(struct request_queue *q)
cancel any pending callbacks on a queue

Parameters

struct request_queue *q
the queue

Description

The block layer may perform asynchronous callback activity on a queue, such as calling the unplug function after a timeout. A block device may call blk_sync_queue to ensure that any such activity is cancelled, thus allowing it to release resources that the callbacks might use. The caller must already have made sure that its ->submit_bio will not re-add plugging prior to calling this function.

This function does not cancel any asynchronous activity arising out of elevator or throttling code. That would require elevator_exit() and blkcg_exit_queue() to be called with queue lock initialized.

void **blk_set_pm_only**(struct request_queue *q)
increment pm_only counter

Parameters

struct request_queue *q
request queue pointer

void **blk_put_queue**(struct request_queue *q)
decrement the request_queue refcount

Parameters

struct request_queue *q
the request_queue structure to decrement the refcount for

Description

Decrements the refcount of the request_queue and free it when the refcount reaches 0.

bool **blk_get_queue**(struct request_queue *q)
increment the request_queue refcount

Parameters

struct request_queue *q
the request_queue structure to increment the refcount for

Description

Increment the refcount of the request_queue kobject.

Context

Any context.

void **submit_bio_noacct**(struct *bio* *bio)
re-submit a bio to the block device layer for I/O

Parameters

struct bio *bio
The bio describing the location in memory and on the device.

Description

This is a version of submit_bio() that shall only be used for I/O that is resubmitted to lower level drivers by stacking block drivers. All file systems and other upper level users of the block layer should use submit_bio() instead.

void **submit_bio**(struct *bio* *bio)
submit a bio to the block device layer for I/O

Parameters

struct bio *bio
The `struct bio` which describes the I/O

Description

submit_bio() is used to submit I/O requests to block devices. It is passed a fully set up `struct bio` that describes the I/O that needs to be done. The bio will be send to the device described by the bi_bdev field.

The success/failure status of the request, along with notification of completion, is delivered asynchronously through the ->bi_end_io() callback in **bio**. The bio must NOT be touched by the caller until ->bi_end_io() has been called.

int **bio_poll**(struct *bio* *bio, struct io_comp_batch *iob, unsigned int flags)
poll for BIO completions

Parameters

struct bio *bio
bio to poll for

struct io_comp_batch *iob
batches of IO

unsigned int flags
BLK_POLL_* flags that control the behavior

Description

Poll for completions on queue associated with the bio. Returns number of completed entries found.

Note

the caller must either be the context that submitted **bio**, or be in a RCU critical section to prevent freeing of **bio**.

unsigned long **bio_start_io_acct**(struct *bio* *bio)
start I/O accounting for bio based drivers

Parameters

struct bio *bio
bio to start account for

Description

Returns the start time that should be passed back to bio_end_io_acct().

int **blk_lld_busy**(struct request_queue *q)
Check if underlying low-level drivers of a device are busy

Parameters

struct request_queue *q
the queue of the device being checked

Description

Check if underlying low-level drivers of a device are busy. If the drivers want to export their busy state, they must set own exporting function using blk_queue_lld_busy() first.

Basically, this function is used only by request stacking drivers to stop dispatching requests to underlying devices when underlying devices are busy. This behavior helps more I/O merging on the queue of the request stacking driver and prevents I/O throughput regression on burst I/O load.

Return

0 - Not busy (The request stacking driver should dispatch request) 1 - Busy (The request stacking driver should stop dispatching request)

void **blk_start_plug**(struct blk_plug *plug)

initialize blk_plug and track it inside the task_struct

Parameters

struct blk_plug *plug
The `struct blk_plug` that needs to be initialized

Description

blk_start_plug() indicates to the block layer an intent by the caller to submit multiple I/O requests in a batch. The block layer may use this hint to defer submitting I/Os from the caller until *blk_finish_plug()* is called. However, the block layer may choose to submit requests before a call to *blk_finish_plug()* if the number of queued I/Os exceeds `BLK_MAX_REQUEST_COUNT`, or if the size of the I/O is larger than `BLK_PLUG_FLUSH_SIZE`. The queued I/Os may also be submitted early if the task schedules (see below).

Tracking blk_plug inside the task_struct will help with auto-flushing the pending I/O should the task end up blocking between *blk_start_plug()* and *blk_finish_plug()*. This is important from a performance perspective, but also ensures that we don't deadlock. For instance, if the task is blocking for a memory allocation, memory reclaim could end up wanting to free a page belonging to that request that is currently residing in our private plug. By flushing the pending I/O when the process goes to sleep, we avoid this kind of deadlock.

void **blk_finish_plug**(struct blk_plug *plug)

mark the end of a batch of submitted I/O

Parameters

struct blk_plug *plug
The `struct blk_plug` passed to *blk_start_plug()*

Description

Indicate that a batch of I/O submissions is complete. This function must be paired with an initial call to *blk_start_plug()*. The intent is to allow the block layer to optimize I/O submission. See the documentation for *blk_start_plug()* for more information.

int **blk_queue_enter**(struct request_queue *q, blk_mq_req_flags_t flags)

try to increase q->q_usage_counter

Parameters

struct request_queue *q
request queue pointer

blk_mq_req_flags_t flags
BLK_MQ_REQ_NOWAIT and/or BLK_MQ_REQ_PM

int **blk_rq_map_user_iov**(struct request_queue *q, struct request *rq, struct rq_map_data *map_data, const struct iov_iter *iter, gfp_t gfp_mask)

map user data to a request, for passthrough requests

Parameters

struct request_queue *q
request queue where request should be inserted

struct request *rq
request to map data to

struct rq_map_data *map_data
pointer to the rq_map_data holding pages (if necessary)

const struct iov_iter *iter
iovec iterator

gfp_t gfp_mask
memory allocation flags

Description

Data will be mapped directly for zero copy I/O, if possible. Otherwise a kernel bounce buffer is used.

A matching *blk_rq_unmap_user()* must be issued at the end of I/O, while still in process context.

int **blk_rq_unmap_user**(struct *bio* *bio)

unmap a request with user data

Parameters

struct bio *bio
start of bio list

Description

Unmap a rq previously mapped by blk_rq_map_user(). The caller must supply the original rq->bio from the blk_rq_map_user() return, since the I/O completion may have changed rq->bio.

int **blk_rq_map_kern**(struct request_queue *q, struct request *rq, void *kbuf, unsigned int len, gfp_t gfp_mask)

map kernel data to a request, for passthrough requests

Parameters

struct request_queue *q
request queue where request should be inserted

struct request *rq
request to fill

void *kbuf
the kernel buffer

unsigned int len
    length of user data

gfp_t gfp_mask
    memory allocation flags

Description

Data will be mapped directly if possible. Otherwise a bounce buffer is used. Can be called multiple times to append multiple buffers.

int **blk_register_queue**(struct gendisk *disk)
    register a block layer queue with sysfs

Parameters

struct gendisk *disk
    Disk of which the request queue should be registered with sysfs.

void **blk_unregister_queue**(struct gendisk *disk)
    counterpart of *blk_register_queue()*

Parameters

struct gendisk *disk
    Disk of which the request queue should be unregistered from sysfs.

Note

the caller is responsible for guaranteeing that this function is called after *blk_register_queue()* has finished.

void **blk_set_stacking_limits**(struct queue_limits *lim)
    set default limits for stacking devices

Parameters

struct queue_limits *lim
    the queue_limits structure to reset

Description

Returns a queue_limit struct to its default state. Should be used by stacking drivers like DM that have no internal limits.

void **blk_queue_bounce_limit**(struct request_queue *q, enum blk_bounce bounce)
    set bounce buffer limit for queue

Parameters

struct request_queue *q
    the request queue for the device

enum blk_bounce bounce
    bounce limit to enforce

Description

Force bouncing for ISA DMA ranges or highmem.

DEPRECATED, don’t use in new code.

void **blk_queue_max_hw_sectors**(struct request_queue *q, unsigned int max_hw_sectors)
set max sectors for a request for this queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_hw_sectors
max hardware sectors in the usual 512b unit

Description

Enables a low level driver to set a hard upper limit, max_hw_sectors, on the size of requests. max_hw_sectors is set by the device driver based upon the capabilities of the I/O controller.

max_dev_sectors is a hard limit imposed by the storage device for READ/WRITE requests. It is set by the disk driver.

max_sectors is a soft limit imposed by the block layer for filesystem type requests. This value can be overridden on a per-device basis in /sys/block/<device>/queue/max_sectors_kb. The soft limit can not exceed max_hw_sectors.

void **blk_queue_chunk_sectors**(struct request_queue *q, unsigned int chunk_sectors)
set size of the chunk for this queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int chunk_sectors
chunk sectors in the usual 512b unit

Description

If a driver doesn't want IOs to cross a given chunk size, it can set this limit and prevent merging across chunks. Note that the block layer must accept a page worth of data at any offset. So if the crossing of chunks is a hard limitation in the driver, it must still be prepared to split single page bios.

void **blk_queue_max_discard_sectors**(struct request_queue *q, unsigned int max_discard_sectors)
set max sectors for a single discard

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_discard_sectors
maximum number of sectors to discard

void **blk_queue_max_secure_erase_sectors**(struct request_queue *q, unsigned int max_sectors)
set max sectors for a secure erase

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_sectors
maximum number of sectors to secure_erase

void **blk_queue_max_write_zeroes_sectors**(struct request_queue *q, unsigned int max_write_zeroes_sectors)

set max sectors for a single write zeroes

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_write_zeroes_sectors
maximum number of sectors to write per command

void **blk_queue_max_zone_append_sectors**(struct request_queue *q, unsigned int max_zone_append_sectors)

set max sectors for a single zone append

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_zone_append_sectors
maximum number of sectors to write per command

void **blk_queue_max_segments**(struct request_queue *q, unsigned short max_segments)

set max hw segments for a request for this queue

Parameters

struct request_queue *q
the request queue for the device

unsigned short max_segments
max number of segments

Description

Enables a low level driver to set an upper limit on the number of hw data segments in a request.

void **blk_queue_max_discard_segments**(struct request_queue *q, unsigned short max_segments)

set max segments for discard requests

Parameters

struct request_queue *q
the request queue for the device

unsigned short max_segments
max number of segments

Description

Enables a low level driver to set an upper limit on the number of segments in a discard request.

void **blk_queue_max_segment_size**(struct request_queue *q, unsigned int max_size)

set max segment size for blk_rq_map_sg

Parameters

struct request_queue *q
the request queue for the device

unsigned int max_size
max size of segment in bytes

Description

Enables a low level driver to set an upper limit on the size of a coalesced segment

void **blk_queue_logical_block_size**(struct request_queue *q, unsigned int size)

set logical block size for the queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int size
the logical block size, in bytes

Description

This should be set to the lowest possible block size that the storage device can address. The default of 512 covers most hardware.

void **blk_queue_physical_block_size**(struct request_queue *q, unsigned int size)

set physical block size for the queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int size
the physical block size, in bytes

Description

This should be set to the lowest possible sector size that the hardware can operate on without reverting to read-modify-write operations.

void **blk_queue_zone_write_granularity**(struct request_queue *q, unsigned int size)

set zone write granularity for the queue

Parameters

struct request_queue *q
the request queue for the zoned device

unsigned int size
the zone write granularity size, in bytes

Description

This should be set to the lowest possible size allowing to write in sequential zones of a zoned block device.

void **blk_queue_alignment_offset**(struct request_queue *q, unsigned int offset)

set physical block alignment offset

Parameters

struct request_queue *q
the request queue for the device

unsigned int offset
alignment offset in bytes

Description

Some devices are naturally misaligned to compensate for things like the legacy DOS partition table 63-sector offset. Low-level drivers should call this function for devices whose first sector is not naturally aligned.

void **blk_limits_io_min**(struct queue_limits *limits, unsigned int min)

set minimum request size for a device

Parameters

struct queue_limits *limits
the queue limits

unsigned int min
smallest I/O size in bytes

Description

Some devices have an internal block size bigger than the reported hardware sector size. This function can be used to signal the smallest I/O the device can perform without incurring a performance penalty.

void **blk_queue_io_min**(struct request_queue *q, unsigned int min)

set minimum request size for the queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int min
smallest I/O size in bytes

Description

Storage devices may report a granularity or preferred minimum I/O size which is the smallest request the device can perform without incurring a performance penalty. For disk drives this is often the physical block size. For RAID arrays it is often the stripe chunk size. A properly aligned multiple of minimum_io_size is the preferred request size for workloads where a high number of I/O operations is desired.

void **blk_limits_io_opt**(struct queue_limits *limits, unsigned int opt)

set optimal request size for a device

Parameters

struct queue_limits *limits
the queue limits

unsigned int opt
smallest I/O size in bytes

Description

Storage devices may report an optimal I/O size, which is the device's preferred unit for sustained I/O. This is rarely reported for disk drives. For RAID arrays it is usually the stripe width or the internal track size. A properly aligned multiple of optimal_io_size is the preferred request size for workloads where sustained throughput is desired.

void **blk_queue_io_opt**(struct request_queue *q, unsigned int opt)
set optimal request size for the queue

Parameters

struct request_queue *q
the request queue for the device

unsigned int opt
optimal request size in bytes

Description

Storage devices may report an optimal I/O size, which is the device's preferred unit for sustained I/O. This is rarely reported for disk drives. For RAID arrays it is usually the stripe width or the internal track size. A properly aligned multiple of optimal_io_size is the preferred request size for workloads where sustained throughput is desired.

int **blk_stack_limits**(struct queue_limits *t, struct queue_limits *b, sector_t start)
adjust queue_limits for stacked devices

Parameters

struct queue_limits *t
the stacking driver limits (top device)

struct queue_limits *b
the underlying queue limits (bottom, component device)

sector_t start
first data sector within component device

Description

This function is used by stacking drivers like MD and DM to ensure that all component devices have compatible block sizes and alignments. The stacking driver must provide a queue_limits struct (top) and then iteratively call the stacking function for all component (bottom) devices. The stacking function will attempt to combine the values and ensure proper alignment.

Returns 0 if the top and bottom queue_limits are compatible. The top device's block sizes and alignment offsets may be adjusted to ensure alignment with the bottom device. If no compatible sizes and alignments exist, -1 is returned and the resulting top queue_limits will have the misaligned flag set to indicate that the alignment_offset is undefined.

void **disk_stack_limits**(struct gendisk *disk, struct block_device *bdev, sector_t offset)
adjust queue limits for stacked drivers

Parameters

struct gendisk *disk
MD/DM gendisk (top)

struct block_device *bdev
the underlying block device (bottom)

sector_t offset
offset to beginning of data within component device

Description

Merges the limits for a top level gendisk and a bottom level block_device.

void **blk_queue_update_dma_pad**(struct request_queue *q, unsigned int mask)
update pad mask

Parameters

struct request_queue *q
the request queue for the device

unsigned int mask
pad mask

Description

Update dma pad mask.

Appending pad buffer to a request modifies the last entry of a scatter list such that it includes the pad buffer.

void **blk_queue_segment_boundary**(struct request_queue *q, unsigned long mask)
set boundary rules for segment merging

Parameters

struct request_queue *q
the request queue for the device

unsigned long mask
the memory boundary mask

void **blk_queue_virt_boundary**(struct request_queue *q, unsigned long mask)
set boundary rules for bio merging

Parameters

struct request_queue *q
the request queue for the device

unsigned long mask
the memory boundary mask

void **blk_queue_dma_alignment**(struct request_queue *q, int mask)
set dma length and memory alignment

Parameters

struct request_queue *q
the request queue for the device

int mask
alignment mask

Description

set required memory and length alignment for direct dma transactions. this is used when building direct io requests for the queue.

void **blk_queue_update_dma_alignment**(struct request_queue *q, int mask)
update dma length and memory alignment

Parameters

struct request_queue *q
the request queue for the device

int mask
alignment mask

Description

update required memory and length alignment for direct dma transactions. If the requested alignment is larger than the current alignment, then the current queue alignment is updated to the new value, otherwise it is left alone. The design of this is to allow multiple objects (driver, device, transport etc) to set their respective alignments without having them interfere.

void **blk_set_queue_depth**(struct request_queue *q, unsigned int depth)
tell the block layer about the device queue depth

Parameters

struct request_queue *q
the request queue for the device

unsigned int depth
queue depth

void **blk_queue_write_cache**(struct request_queue *q, bool wc, bool fua)
configure queue's write cache

Parameters

struct request_queue *q
the request queue for the device

bool wc
write back cache on or off

bool fua
device supports FUA writes, if true

Description

Tell the block layer about the write cache of **q**.

void **blk_queue_required_elevator_features**(struct request_queue *q, unsigned int features)

Set a queue required elevator features

Parameters

struct request_queue *q
the request queue for the target device

unsigned int features
Required elevator features OR'ed together

Description

Tell the block layer that for the device controlled through **q**, only the only elevators that can be used are those that implement at least the set of features specified by **features**.

bool **blk_queue_can_use_dma_map_merging**(struct request_queue *q, struct device *dev)

configure queue for merging segments.

Parameters

struct request_queue *q
the request queue for the device

struct device *dev
the device pointer for dma

Description

Tell the block layer about merging the segments by dma map of **q**.

void **disk_set_zoned**(struct gendisk *disk, enum blk_zoned_model model)

configure the zoned model for a disk

Parameters

struct gendisk *disk
the gendisk of the queue to configure

enum blk_zoned_model model
the zoned model to set

Description

Set the zoned model of **disk** to **model**.

When **model** is BLK_ZONED_HM (host managed), this should be called only if zoned block device support is enabled (CONFIG_BLK_DEV_ZONED option). If **model** specifies BLK_ZONED_HA (host aware), the effective model used depends on CONFIG_BLK_DEV_ZONED settings and on the existence of partitions on the disk.

int **blkdev_issue_flush**(struct block_device *bdev)

queue a flush

Parameters

struct block_device *bdev
blockdev to issue flush for

Description

Issue a flush for the block device in question.

int **blkdev_issue_discard**(struct block_device *bdev, sector_t sector, sector_t nr_sects, gfp_t gfp_mask)

queue a discard

Parameters

struct block_device *bdev
blockdev to issue discard for

sector_t sector
start sector

sector_t nr_sects
number of sectors to discard

gfp_t gfp_mask
memory allocation flags (for bio_alloc)

Description

Issue a discard request for the sectors in question.

int **__blkdev_issue_zeroout**(struct block_device *bdev, sector_t sector, sector_t nr_sects, gfp_t gfp_mask, struct bio **biop, unsigned flags)

generate number of zero filed write bios

Parameters

struct block_device *bdev
blockdev to issue

sector_t sector
start sector

sector_t nr_sects
number of sectors to write

gfp_t gfp_mask
memory allocation flags (for bio_alloc)

struct bio **biop
pointer to anchor bio

unsigned flags
controls detailed behavior

Description

Zero-fill a block range, either using hardware offload or by explicitly writing zeroes to the device.

If a device is using logical block provisioning, the underlying space will not be released if `flags` contains BLKDEV_ZERO_NOUNMAP.

If `flags` contains BLKDEV_ZERO_NOFALLBACK, the function will return -EOPNOTSUPP if no explicit hardware offload for zeroing is provided.

int **blkdev_issue_zeroout**(struct block_device *bdev, sector_t sector, sector_t nr_sects, gfp_t gfp_mask, unsigned flags)

zero-fill a block range

Parameters

struct block_device *bdev
blockdev to write

sector_t sector
start sector

sector_t nr_sects
number of sectors to write

gfp_t gfp_mask
memory allocation flags (for bio_alloc)

unsigned flags
controls detailed behavior

Description

Zero-fill a block range, either using hardware offload or by explicitly writing zeroes to the device. See *__blkdev_issue_zeroout()* for the valid values for `flags`.

int **blk_rq_count_integrity_sg**(struct request_queue *q, struct *bio* *bio)

Count number of integrity scatterlist elements

Parameters

struct request_queue *q
request queue

struct bio *bio
bio with integrity metadata attached

Description

Returns the number of elements required in a scatterlist corresponding to the integrity metadata in a bio.

int **blk_rq_map_integrity_sg**(struct request_queue *q, struct *bio* *bio, struct scatterlist *sglist)

Map integrity metadata into a scatterlist

Parameters

struct request_queue *q
request queue

struct bio *bio
bio with integrity metadata attached

struct scatterlist *sglist
target scatterlist

Description

Map the integrity vectors in request into a scatterlist. The scatterlist must be big enough to hold all elements. I.e. sized using *blk_rq_count_integrity_sg()*.

int **blk_integrity_compare**(struct gendisk *gd1, struct gendisk *gd2)

Compare integrity profile of two disks

Parameters

struct gendisk *gd1
Disk to compare

struct gendisk *gd2
Disk to compare

Description

Meta-devices like DM and MD need to verify that all sub-devices use the same integrity format before advertising to upper layers that they can send/receive integrity metadata. This function can be used to check whether two gendisk devices have compatible integrity formats.

void **blk_integrity_register**(struct gendisk *disk, struct blk_integrity *template)

Register a gendisk as being integrity-capable

Parameters

struct gendisk *disk
struct gendisk pointer to make integrity-aware

struct blk_integrity *template
block integrity profile to register

Description

When a device needs to advertise itself as being able to send/receive integrity metadata it must use this function to register the capability with the block layer. The template is a blk_integrity struct with values appropriate for the underlying hardware. See Documentation/block/data-integrity.rst.

void **blk_integrity_unregister**(struct gendisk *disk)

Unregister block integrity profile

Parameters

struct gendisk *disk
disk whose integrity profile to unregister

Description

This function unregisters the integrity capability from a block device.

int **blk_trace_ioctl**(struct block_device *bdev, unsigned cmd, char __user *arg)

handle the ioctls associated with tracing

Parameters

struct block_device *bdev
the block device

unsigned cmd
the ioctl cmd

char __user *arg
the argument data, if any

void **blk_trace_shutdown**(struct request_queue *q)

stop and cleanup trace structures

Parameters

struct request_queue *q
the request queue associated with the device

void **blk_add_trace_rq**(struct request *rq, blk_status_t error, unsigned int nr_bytes, u32 what, u64 cgid)

Add a trace for a request oriented action

Parameters

struct request *rq
the source request

blk_status_t error
return status to log

unsigned int nr_bytes
number of completed bytes

u32 what
the action

u64 cgid
the cgroup info

Description

Records an action against a request. Will log the bio offset + size.

void **blk_add_trace_bio**(struct request_queue *q, struct *bio* *bio, u32 what, int error)

Add a trace for a bio oriented action

Parameters

struct request_queue *q
queue the io is for

struct bio *bio
the source bio

u32 what
the action

int error
error, if any

Description

Records an action against a bio. Will log the bio offset + size.

void **blk_add_trace_bio_remap**(void *ignore, struct *bio* *bio, dev_t dev, sector_t from)

Add a trace for a bio-remap operation

Parameters

void *ignore
trace callback data parameter (not used)

struct bio *bio
the source bio

dev_t dev
source device

sector_t from
source sector

Description

Called after a bio is remapped to a different device and/or sector.

void **blk_add_trace_rq_remap**(void *ignore, struct request *rq, dev_t dev, sector_t from)
Add a trace for a request-remap operation

Parameters

void *ignore
trace callback data parameter (not used)

struct request *rq
the source request

dev_t dev
target device

sector_t from
source sector

Description

Device mapper remaps request to other devices. Add a trace for that action.

void **disk_release**(struct device *dev)
releases all allocated resources of the gendisk

Parameters

struct device *dev
the device representing this disk

Description

This function releases all allocated resources of the gendisk.

Drivers which used __device_add_disk() have a gendisk with a request_queue assigned. Since the request_queue sits on top of the gendisk for these drivers we also call blk_put_queue() for them, and we expect the request_queue refcount to reach 0 at this point, and so the request_queue will also be freed prior to the disk.

Context

can sleep

int **__register_blkdev**(unsigned int major, const char *name, void (*probe)(dev_t devt))
register a new block device

Parameters

unsigned int major
the requested major device number [1..BLKDEV_MAJOR_MAX-1]. If **major** = 0, try to allocate any unused major number.

const char *name
the name of the new block device as a zero terminated string

void (*probe)(dev_t devt)
pre-devtmpfs / pre-udev callback used to create disks when their pre-created device node is accessed. When a probe call uses add_disk() and it fails the driver must cleanup resources. This interface may soon be removed.

Description

The **name** must be unique within the system.

The return value depends on the **major** input parameter:

- if a major device number was requested in range [1..BLKDEV_MAJOR_MAX-1] then the function returns zero on success, or a negative error code
- if any unused major number was requested with **major** = 0 parameter then the return value is the allocated major number in range [1..BLKDEV_MAJOR_MAX-1] or a negative error code otherwise

See Documentation/admin-guide/devices.txt for the list of allocated major numbers.

Use register_blkdev instead for any new code.

int **device_add_disk**(struct device *parent, struct gendisk *disk, const struct attribute_group **groups)
add disk information to kernel list

Parameters

struct device *parent
parent device for the disk

struct gendisk *disk
per-device partitioning information

const struct attribute_group **groups
Additional per-device sysfs groups

Description

This function registers the partitioning information in **disk** with the kernel.

void **blk_mark_disk_dead**(struct gendisk *disk)
mark a disk as dead

Parameters

struct gendisk *disk
disk to mark as dead

Description

Mark as disk as dead (e.g. surprise removed) and don't accept any new I/O to this disk.

void **del_gendisk**(struct gendisk *disk)
remove the gendisk

Parameters

struct gendisk *disk
the struct gendisk to remove

Description

Removes the gendisk and all its associated resources. This deletes the partitions associated with the gendisk, and unregisters the associated request_queue.

This is the counter to the respective __device_add_disk() call.

The final removal of the struct gendisk happens when its refcount reaches 0 with *put_disk()*, which should be called after *del_gendisk()*, if __device_add_disk() was used.

Drivers exist which depend on the release of the gendisk to be synchronous, it should not be deferred.

Context

can sleep

void **invalidate_disk**(struct gendisk *disk)
invalidate the disk

Parameters

struct gendisk *disk
the struct gendisk to invalidate

Description

A helper to invalidates the disk. It will clean the disk's associated buffer/page caches and reset its internal states so that the disk can be reused by the drivers.

Context

can sleep

void **put_disk**(struct gendisk *disk)
decrements the gendisk refcount

Parameters

struct gendisk *disk
the struct gendisk to decrement the refcount for

Description

This decrements the refcount for the struct gendisk. When this reaches 0 we'll have *disk_release()* called.

Note

for blk-mq disk put_disk must be called before freeing the tag_set when handling probe errors (that is before add_disk() is called).

Context

Any context, but the last reference must not be dropped from atomic context.

void **set_disk_ro**(struct gendisk *disk, bool read_only)

set a gendisk read-only

Parameters

struct gendisk *disk
gendisk to operate on

bool read_only
`true` to set the disk read-only, `false` set the disk read/write

Description

This function is used to indicate whether a given disk device should have its read-only flag set. *set_disk_ro()* is typically used by device drivers to indicate whether the underlying physical device is write-protected.

int **freeze_bdev**(struct block_device *bdev)

lock a filesystem and force it into a consistent state

Parameters

struct block_device *bdev
blockdevice to lock

Description

If a superblock is found on this device, we take the s_umount semaphore on it to make sure nobody unmounts until the snapshot creation is done. The reference counter (bd_fsfreeze_count) guarantees that only the last unfreeze process can unfreeze the frozen filesystem actually when multiple freeze requests arrive simultaneously. It counts up in *freeze_bdev()* and count down in *thaw_bdev()*. When it becomes 0, *thaw_bdev()* will unfreeze actually.

int **thaw_bdev**(struct block_device *bdev)

unlock filesystem

Parameters

struct block_device *bdev
blockdevice to unlock

Description

Unlocks the filesystem and marks it writeable again after *freeze_bdev()*.

int **bd_prepare_to_claim**(struct block_device *bdev, void *holder, const struct blk_holder_ops *hops)

claim a block device

Parameters

struct block_device *bdev
block device of interest

void *holder
holder trying to claim **bdev**

const struct blk_holder_ops *hops
holder ops.

Description

Claim **bdev**. This function fails if **bdev** is already claimed by another holder and waits if another claiming is in progress. return, the caller has ownership of bd_claiming and bd_holder[s].

Return

0 if **bdev** can be claimed, -EBUSY otherwise.

void **bd_abort_claiming**(struct block_device *bdev, void *holder)
abort claiming of a block device

Parameters

struct block_device *bdev
block device of interest

void *holder
holder that has claimed **bdev**

Description

Abort claiming of a block device when the exclusive open failed. This can be also used when exclusive open is not actually desired and we just needed to block other exclusive openers for a while.

struct block_device ***blkdev_get_by_dev**(dev_t dev, blk_mode_t mode, void *holder, const struct blk_holder_ops *hops)
open a block device by device number

Parameters

dev_t dev
device number of block device to open

blk_mode_t mode
open mode (BLK_OPEN_*)

void *holder
exclusive holder identifier

const struct blk_holder_ops *hops
holder operations

Description

Open the block device described by device number **dev**. If **holder** is not `NULL`, the block device is opened with exclusive access. Exclusive opens may nest for the same **holder**.

Use this interface ONLY if you really do not have anything better - i.e. when you are behind a truly sucky interface and all you are given is a device number. Everything else should use *blkdev_get_by_path()*.

Context

Might sleep.

Return

Reference to the block_device on success, ERR_PTR(-errno) on failure.

struct block_device ***blkdev_get_by_path**(const char *path, blk_mode_t mode, void *holder, const struct blk_holder_ops *hops)

open a block device by name

Parameters

const char *path
path to the block device to open

blk_mode_t mode
open mode (BLK_OPEN_*)

void *holder
exclusive holder identifier

const struct blk_holder_ops *hops
holder operations

Description

Open the block device described by the device file at **path**. If **holder** is not `NULL`, the block device is opened with exclusive access. Exclusive opens may nest for the same **holder**.

Context

Might sleep.

Return

Reference to the block_device on success, ERR_PTR(-errno) on failure.

int **lookup_bdev**(const char *pathname, dev_t *dev)

Look up a struct block_device by name.

Parameters

const char *pathname
Name of the block device in the filesystem.

dev_t *dev
Pointer to the block device's dev_t, if found.

Description

Lookup the block device's dev_t at **pathname** in the current namespace if possible and return it in **dev**.

Context

May sleep.

Return

0 if succeeded, negative errno otherwise.

void **bdev_mark_dead**(struct block_device *bdev, bool surprise)

mark a block device as dead

Parameters

struct block_device *bdev
block device to operate on

bool surprise
indicate a surprise removal

Description

Tell the file system that this devices or media is dead. If **surprise** is set to `true` the device or media is already gone, if not we are preparing for an orderly removal.

This calls into the file system, which then typicall syncs out all dirty data and writes back inodes and then invalidates any cached data in the inodes on the file system. In addition we also invalidate the block device mapping.

1.1.14 Char devices

int **register_chrdev_region**(dev_t from, unsigned count, const char *name)
register a range of device numbers

Parameters

dev_t from
the first in the desired range of device numbers; must include the major number.

unsigned count
the number of consecutive device numbers required

const char *name
the name of the device or driver.

Description

Return value is zero on success, a negative error code on failure.

int **alloc_chrdev_region**(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
register a range of char device numbers

Parameters

dev_t *dev
output parameter for first assigned number

unsigned baseminor
first of the requested range of minor numbers

unsigned count
the number of minor numbers required

const char *name
the name of the associated device or driver

Description

Allocates a range of char device numbers. The major number will be chosen dynamically, and returned (along with the first minor number) in **dev**. Returns zero or a negative error code.

int **__register_chrdev**(unsigned int major, unsigned int baseminor, unsigned int count, const char *name, const struct file_operations *fops)
create and register a cdev occupying a range of minors

Parameters

unsigned int major
major device number or 0 for dynamic allocation

unsigned int baseminor
first of the requested range of minor numbers

unsigned int count
the number of minor numbers required

const char *name
name of this range of devices

const struct file_operations *fops
file operations associated with this devices

Description

If **major** == 0 this functions will dynamically allocate a major and return its number.

If **major** > 0 this function will attempt to reserve a device with the given major number and will return zero on success.

Returns a -ve errno on failure.

The name of this device has nothing to do with the name of the device in /dev. It only helps to keep track of the different owners of devices. If your module name has only one type of devices it's ok to use e.g. the name of the module here.

void **unregister_chrdev_region**(dev_t from, unsigned count)
unregister a range of device numbers

Parameters

dev_t from
the first in the range of numbers to unregister

unsigned count
the number of device numbers to unregister

Description

This function will unregister a range of **count** device numbers, starting with **from**. The caller should normally be the one who allocated those numbers in the first place...

void **__unregister_chrdev**(unsigned int major, unsigned int baseminor, unsigned int count, const char *name)
unregister and destroy a cdev

Parameters

unsigned int major
major device number

unsigned int baseminor
first of the range of minor numbers

unsigned int count
the number of minor numbers this cdev is occupying

const char *name
name of this range of devices

Description

Unregister and destroy the cdev occupying the region described by **major**, **baseminor** and **count**. This function undoes what *__register_chrdev()* did.

int **cdev_add**(struct cdev *p, dev_t dev, unsigned count)
add a char device to the system

Parameters

struct cdev *p
the cdev structure for the device

dev_t dev
the first device number for which this device is responsible

unsigned count
the number of consecutive minor numbers corresponding to this device

Description

cdev_add() adds the device represented by **p** to the system, making it live immediately. A negative error code is returned on failure.

void **cdev_set_parent**(struct cdev *p, struct kobject *kobj)
set the parent kobject for a char device

Parameters

struct cdev *p
the cdev structure

struct kobject *kobj
the kobject to take a reference to

Description

cdev_set_parent() sets a parent kobject which will be referenced appropriately so the parent is not freed before the cdev. This should be called before cdev_add.

int **cdev_device_add**(struct *cdev* *cdev, struct device *dev)
add a char device and it's corresponding struct device, linkink

Parameters

struct cdev *cdev
the cdev structure

struct device *dev
the device structure

Description

cdev_device_add() adds the char device represented by **cdev** to the system, just as cdev_add does. It then adds **dev** to the system using device_add The dev_t for the char device will be taken from the struct device which needs to be initialized first. This helper function correctly takes a reference to the parent device so the parent will not get released until all references to the cdev are released.

This helper uses dev->devt for the device number. If it is not set it will not add the cdev and it will be equivalent to device_add.

This function should be used whenever the struct cdev and the struct device are members of the same structure whose lifetime is managed by the struct device.

NOTE

Callers must assume that userspace was able to open the cdev and can call cdev fops callbacks at any time, even if this function fails.

void **cdev_device_del**(struct *cdev* *cdev, struct device *dev)

inverse of cdev_device_add

Parameters

struct cdev *cdev
the cdev structure

struct device *dev
the device structure

Description

cdev_device_del() is a helper function to call cdev_del and device_del. It should be used whenever cdev_device_add is used.

If dev->devt is not set it will not remove the cdev and will be equivalent to device_del.

NOTE

This guarantees that associated sysfs callbacks are not running or runnable, however any cdevs already open will remain and their fops will still be callable even after this function returns.

void **cdev_del**(struct cdev *p)

remove a cdev from the system

Parameters

struct cdev *p
the cdev structure to be removed

Description

cdev_del() removes **p** from the system, possibly freeing the structure itself.

NOTE

This guarantees that cdev device will no longer be able to be opened, however any cdevs already open will remain and their fops will still be callable even after cdev_del returns.

struct cdev ***cdev_alloc**(void)

allocate a cdev structure

Parameters

void
no arguments

Description

Allocates and returns a cdev structure, or NULL on failure.

void **cdev_init**(struct *cdev* *cdev, const struct file_operations *fops)
initialize a cdev structure

Parameters

struct cdev *cdev
the structure to initialize

const struct file_operations *fops
the file_operations for this device

Description

Initializes **cdev**, remembering **fops**, making it ready to add to the system with *cdev_add()*.

1.1.15 Clock Framework

The clock framework defines programming interfaces to support software management of the system clock tree. This framework is widely used with System-On-Chip (SOC) platforms to support power management and various devices which may need custom clock rates. Note that these "clocks" don't relate to timekeeping or real time clocks (RTCs), each of which have separate frameworks. These `struct clk` instances may be used to manage for example a 96 MHz signal that is used to shift bits into and out of peripherals or busses, or otherwise trigger synchronous state machine transitions in system hardware.

Power management is supported by explicit software clock gating: unused clocks are disabled, so the system doesn't waste power changing the state of transistors that aren't in active use. On some systems this may be backed by hardware clock gating, where clocks are gated without being disabled in software. Sections of chips that are powered but not clocked may be able to retain their last state. This low power state is often called a *retention mode*. This mode still incurs leakage currents, especially with finer circuit geometries, but for CMOS circuits power is mostly used by clocked state changes.

Power-aware drivers only enable their clocks when the device they manage is in active use. Also, system sleep states often differ according to which clock domains are active: while a "standby" state may allow wakeup from several active domains, a "mem" (suspend-to-RAM) state may require a more wholesale shutdown of clocks derived from higher speed PLLs and oscillators, limiting the number of possible wakeup event sources. A driver's suspend method may need to be aware of system-specific clock constraints on the target sleep state.

Some platforms support programmable clock generators. These can be used by external chips of various kinds, such as other CPUs, multimedia codecs, and devices with strict requirements for interface clocking.

struct **clk_notifier**
associate a clk with a notifier

Definition:

```
struct clk_notifier {
    struct clk                      *clk;
    struct srcu_notifier_head       notifier_head;
    struct list_head                node;
};
```

Members

clk
struct clk * to associate the notifier with

notifier_head
a blocking_notifier_head for this clk

node
linked list pointers

Description

A list of *struct clk_notifier* is maintained by the notifier code. An entry is created whenever code registers the first notifier on a particular **clk**. Future notifiers on that **clk** are added to the **notifier_head**.

struct **clk_notifier_data**
rate data to pass to the notifier callback

Definition:

```
struct clk_notifier_data {
    struct clk              *clk;
    unsigned long           old_rate;
    unsigned long           new_rate;
};
```

Members

clk
struct clk * being changed

old_rate
previous rate of this clk

new_rate
new rate of this clk

Description

For a pre-notifier, old_rate is the clk's rate before this rate change, and new_rate is what the rate will be in the future. For a post-notifier, old_rate and new_rate are both set to the clk's current rate (this was done to optimize the implementation).

struct **clk_bulk_data**
Data used for bulk clk operations.

Definition:

```
struct clk_bulk_data {
    const char              *id;
    struct clk              *clk;
};
```

Members

id
clock consumer ID

clk
struct clk * to store the associated clock

Description

The CLK APIs provide a series of clk_bulk_() API calls as a convenience to consumers which require multiple clks. This structure is used to manage data for these calls.

int **clk_notifier_register**(struct *clk* *clk, struct notifier_block *nb)
register a clock rate-change notifier callback

Parameters

struct clk *clk
clock whose rate we are interested in

struct notifier_block *nb
notifier block with callback function pointer

Description

ProTip: debugging across notifier chains can be frustrating. Make sure that your notifier callback function prints a nice big warning in case of failure.

int **clk_notifier_unregister**(struct *clk* *clk, struct notifier_block *nb)
unregister a clock rate-change notifier callback

Parameters

struct clk *clk
clock whose rate we are no longer interested in

struct notifier_block *nb
notifier block which will be unregistered

int **devm_clk_notifier_register**(struct device *dev, struct *clk* *clk, struct notifier_block *nb)
register a managed rate-change notifier callback

Parameters

struct device *dev
device for clock "consumer"

struct clk *clk
clock whose rate we are interested in

struct notifier_block *nb
notifier block with callback function pointer

Description

Returns 0 on success, -EERROR otherwise

long **clk_get_accuracy**(struct *clk* *clk)
obtain the clock accuracy in ppb (parts per billion) for a clock source.

Parameters

struct clk *clk
clock source

Description

This gets the clock source accuracy expressed in ppb. A perfect clock returns 0.

int **clk_set_phase**(struct *clk* *clk, int degrees)

adjust the phase shift of a clock signal

Parameters

struct clk *clk
clock signal source

int degrees
number of degrees the signal is shifted

Description

Shifts the phase of a clock signal by the specified degrees. Returns 0 on success, -EERROR otherwise.

int **clk_get_phase**(struct *clk* *clk)

return the phase shift of a clock signal

Parameters

struct clk *clk
clock signal source

Description

Returns the phase shift of a clock node in degrees, otherwise returns -EERROR.

int **clk_set_duty_cycle**(struct *clk* *clk, unsigned int num, unsigned int den)

adjust the duty cycle ratio of a clock signal

Parameters

struct clk *clk
clock signal source

unsigned int num
numerator of the duty cycle ratio to be applied

unsigned int den
denominator of the duty cycle ratio to be applied

Description

Adjust the duty cycle of a clock signal by the specified ratio. Returns 0 on success, -EERROR otherwise.

int **clk_get_scaled_duty_cycle**(struct *clk* *clk, unsigned int scale)

return the duty cycle ratio of a clock signal

Parameters

struct clk *clk
clock signal source

unsigned int scale
scaling factor to be applied to represent the ratio as an integer

Description

Returns the duty cycle ratio multiplied by the scale provided, otherwise returns -EERROR.

bool **clk_is_match**(const struct clk *p, const struct clk *q)
check if two clk's point to the same hardware clock

Parameters

const struct clk *p
clk compared against q

const struct clk *q
clk compared against p

Description

Returns true if the two struct clk pointers both point to the same hardware clock node. Put differently, returns true if **p** and **q** share the same `struct clk_core` object.

Returns false otherwise. Note that two NULL clks are treated as matching.

int **clk_rate_exclusive_get**(struct *clk* *clk)
get exclusivity over the rate control of a producer

Parameters

struct clk *clk
clock source

Description

This function allows drivers to get exclusive control over the rate of a provider. It prevents any other consumer to execute, even indirectly, opereation which could alter the rate of the provider or cause glitches

If exlusivity is claimed more than once on clock, even by the same driver, the rate effectively gets locked as exclusivity can't be preempted.

Must not be called from within atomic context.

Returns success (0) or negative errno.

void **clk_rate_exclusive_put**(struct *clk* *clk)
release exclusivity over the rate control of a producer

Parameters

struct clk *clk
clock source

Description

This function allows drivers to release the exclusivity it previously got from *clk_rate_exclusive_get()*

The caller must balance the number of *clk_rate_exclusive_get()* and *clk_rate_exclusive_put()* calls.

Must not be called from within atomic context.

int **clk_prepare**(struct *clk* *clk)
prepare a clock source

Parameters

struct clk *clk
clock source

Description

This prepares the clock source for use.

Must not be called from within atomic context.

bool **clk_is_enabled_when_prepared**(struct *clk* *clk)
indicate if preparing a clock also enables it.

Parameters

struct clk *clk
clock source

Description

Returns true if *clk_prepare()* implicitly enables the clock, effectively making clk_enable()/clk_disable() no-ops, false otherwise.

This is of interest mainly to the power management code where actually disabling the clock also requires unpreparing it to have any material effect.

Regardless of the value returned here, the caller must always invoke clk_enable() or clk_prepare_enable() and counterparts for usage counts to be right.

void **clk_unprepare**(struct *clk* *clk)
undo preparation of a clock source

Parameters

struct clk *clk
clock source

Description

This undoes a previously prepared clock. The caller must balance the number of prepare and unprepare calls.

Must not be called from within atomic context.

struct clk ***clk_get**(struct device *dev, const char *id)
lookup and obtain a reference to a clock producer.

Parameters

struct device *dev
device for clock "consumer"

const char *id
clock consumer ID

Description

Returns a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. (IOW, **id** may be identical strings, but clk_get may return different clock producers depending on **dev**.)

Drivers must assume that the clock source is not enabled.

clk_get should not be called from within interrupt context.

int **clk_bulk_get**(struct device *dev, int num_clks, struct *clk_bulk_data* *clks)

lookup and obtain a number of references to clock producer.

Parameters

struct device *dev
device for clock "consumer"

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Description

This helper function allows drivers to get several clk consumers in one operation. If any of the clk cannot be acquired then any clks that were obtained will be freed before returning to the caller.

Returns 0 if all clocks specified in clk_bulk_data table are obtained successfully, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **clk_bulk_data.id** to determine the clock consumer, and thereby the clock producer. The clock returned is stored in each **clk_bulk_data.clk** field.

Drivers must assume that the clock source is not enabled.

clk_bulk_get should not be called from within interrupt context.

int **clk_bulk_get_all**(struct device *dev, struct *clk_bulk_data* **clks)

lookup and obtain all available references to clock producer.

Parameters

struct device *dev
device for clock "consumer"

struct clk_bulk_data **clks
pointer to the clk_bulk_data table of consumer

Description

This helper function allows drivers to get all clk consumers in one operation. If any of the clk cannot be acquired then any clks that were obtained will be freed before returning to the caller.

Returns a positive value for the number of clocks obtained while the clock references are stored in the clk_bulk_data table in **clks** field. Returns 0 if there're none and a negative value if something failed.

Drivers must assume that the clock source is not enabled.

clk_bulk_get should not be called from within interrupt context.

int **clk_bulk_get_optional**(struct device *dev, int num_clks, struct *clk_bulk_data* *clks)
lookup and obtain a number of references to clock producer

Parameters

struct device *dev
device for clock "consumer"

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Description

Behaves the same as *clk_bulk_get()* except where there is no clock producer. In this case, instead of returning -ENOENT, the function returns 0 and NULL for a clk for which a clock producer could not be determined.

int **devm_clk_bulk_get**(struct device *dev, int num_clks, struct *clk_bulk_data* *clks)
managed get multiple clk consumers

Parameters

struct device *dev
device for clock "consumer"

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Description

Return 0 on success, an errno on failure.

This helper function allows drivers to get several clk consumers in one operation with management, the clks will automatically be freed when the device is unbound.

int **devm_clk_bulk_get_optional**(struct device *dev, int num_clks, struct *clk_bulk_data* *clks)
managed get multiple optional consumer clocks

Parameters

struct device *dev
device for clock "consumer"

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
pointer to the clk_bulk_data table of consumer

Description

Behaves the same as devm_clk_bulk_get() except where there is no clock producer. In this case, instead of returning -ENOENT, the function returns NULL for given clk. It is assumed all clocks in clk_bulk_data are optional.

Returns 0 if all clocks specified in clk_bulk_data table are obtained successfully or for any clk there was no clk provider available, otherwise returns valid IS_ERR() condition containing errno. The implementation uses **dev** and **clk_bulk_data.id** to determine the clock consumer, and thereby the clock producer. The clock returned is stored in each **clk_bulk_data.clk** field.

Drivers must assume that the clock source is not enabled.

clk_bulk_get should not be called from within interrupt context.

int **devm_clk_bulk_get_all**(struct device *dev, struct *clk_bulk_data* **clks)
managed get multiple clk consumers

Parameters

struct device *dev
device for clock "consumer"

struct clk_bulk_data **clks
pointer to the clk_bulk_data table of consumer

Description

Returns a positive value for the number of clocks obtained while the clock references are stored in the clk_bulk_data table in **clks** field. Returns 0 if there're none and a negative value if something failed.

This helper function allows drivers to get several clk consumers in one operation with management, the clks will automatically be freed when the device is unbound.

struct clk ***devm_clk_get**(struct device *dev, const char *id)
lookup and obtain a managed reference to a clock producer.

Parameters

struct device *dev
device for clock "consumer"

const char *id
clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. (IOW, **id** may be identical strings, but clk_get may return different clock producers depending on **dev**.)

Description

Drivers must assume that the clock source is neither prepared nor enabled.

The clock will automatically be freed when the device is unbound from the bus.

struct clk ***devm_clk_get_prepared**(struct device *dev, const char *id)
devm_clk_get() + *clk_prepare()*

Parameters

struct device *dev
device for clock "consumer"

const char *id
clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. (IOW, **id** may be identical strings, but clk_get may return different clock producers depending on **dev**.)

Description

The returned clk (if valid) is prepared. Drivers must however assume that the clock is not enabled.

The clock will automatically be unprepared and freed when the device is unbound from the bus.

struct clk ***devm_clk_get_enabled**(struct device *dev, const char *id)
devm_clk_get() + clk_prepare_enable()

Parameters

struct device *dev
device for clock "consumer"

const char *id
clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. (IOW, **id** may be identical strings, but clk_get may return different clock producers depending on **dev**.)

Description

The returned clk (if valid) is prepared and enabled.

The clock will automatically be disabled, unprepared and freed when the device is unbound from the bus.

struct clk ***devm_clk_get_optional**(struct device *dev, const char *id)
lookup and obtain a managed reference to an optional clock producer.

Parameters

struct device *dev
device for clock "consumer"

const char *id
: clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. If no such clk is found, it returns NULL which serves as a dummy clk. That's the only difference compared to devm_clk_get().

Description

Drivers must assume that the clock source is neither prepared nor enabled.

The clock will automatically be freed when the device is unbound from the bus.

struct clk ***devm_clk_get_optional_prepared**(struct device *dev, const char *id)
: devm_clk_get_optional() + *clk_prepare()*

Parameters

struct device *dev
: device for clock "consumer"

const char *id
: clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. If no such clk is found, it returns NULL which serves as a dummy clk. That's the only difference compared to *devm_clk_get_prepared()*.

Description

The returned clk (if valid) is prepared. Drivers must however assume that the clock is not enabled.

The clock will automatically be unprepared and freed when the device is unbound from the bus.

struct clk ***devm_clk_get_optional_enabled**(struct device *dev, const char *id)
: devm_clk_get_optional() + clk_prepare_enable()

Parameters

struct device *dev
: device for clock "consumer"

const char *id
: clock consumer ID

Context

May sleep.

Return

a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev** and **id** to determine the clock consumer, and thereby the clock producer. If no such clk is found, it returns NULL which serves as a dummy clk. That's the only difference compared to *devm_clk_get_enabled()*.

Description

The returned clk (if valid) is prepared and enabled.

The clock will automatically be disabled, unprepared and freed when the device is unbound from the bus.

struct clk ***devm_get_clk_from_child**(struct device *dev, struct device_node *np, const char *con_id)

lookup and obtain a managed reference to a clock producer from child node.

Parameters

struct device *dev
device for clock "consumer"

struct device_node *np
pointer to clock consumer node

const char *con_id
clock consumer ID

Description

This function parses the clocks, and uses them to look up the struct clk from the registered list of clock providers by using **np** and **con_id**

The clock will automatically be freed when the device is unbound from the bus.

int **clk_enable**(struct *clk* *clk)

inform the system when the clock source should be running.

Parameters

struct clk *clk
clock source

Description

If the clock can not be enabled/disabled, this should return success.

May be called from atomic contexts.

Returns success (0) or negative errno.

int **clk_bulk_enable**(int num_clks, const struct *clk_bulk_data* *clks)

inform the system when the set of clks should be running.

Parameters

int num_clks
the number of clk_bulk_data

const struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Description

May be called from atomic contexts.

Returns success (0) or negative errno.

void **clk_disable**(struct *clk* *clk)

inform the system when the clock source is no longer required.

Parameters

struct clk *clk
clock source

Description

Inform the system that a clock source is no longer required by a driver and may be shut down.

May be called from atomic contexts.

Implementation detail: if the clock source is shared between multiple drivers, clk_enable() calls must be balanced by the same number of clk_disable() calls for the clock source to be disabled.

void **clk_bulk_disable**(int num_clks, const struct *clk_bulk_data* *clks)

inform the system when the set of clks is no longer required.

Parameters

int num_clks
the number of clk_bulk_data

const struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Description

Inform the system that a set of clks is no longer required by a driver and may be shut down.

May be called from atomic contexts.

Implementation detail: if the set of clks is shared between multiple drivers, *clk_bulk_enable()* calls must be balanced by the same number of *clk_bulk_disable()* calls for the clock source to be disabled.

unsigned long **clk_get_rate**(struct *clk* *clk)

obtain the current clock rate (in Hz) for a clock source. This is only valid once the clock source has been enabled.

Parameters

struct clk *clk
clock source

void **clk_put**(struct *clk* *clk)

"free" the clock source

Parameters

struct clk *clk
clock source

Note

drivers must ensure that all clk_enable calls made on this clock source are balanced by clk_disable calls prior to calling this function.

Description

clk_put should not be called from within interrupt context.

void **clk_bulk_put**(int num_clks, struct *clk_bulk_data* *clks)
“free” the clock source

Parameters

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Note

drivers must ensure that all clk_bulk_enable calls made on this clock source are balanced by clk_bulk_disable calls prior to calling this function.

Description

clk_bulk_put should not be called from within interrupt context.

void **clk_bulk_put_all**(int num_clks, struct *clk_bulk_data* *clks)
“free” all the clock source

Parameters

int num_clks
the number of clk_bulk_data

struct clk_bulk_data *clks
the clk_bulk_data table of consumer

Note

drivers must ensure that all clk_bulk_enable calls made on this clock source are balanced by clk_bulk_disable calls prior to calling this function.

Description

clk_bulk_put_all should not be called from within interrupt context.

void **devm_clk_put**(struct device *dev, struct *clk* *clk)
“free” a managed clock source

Parameters

struct device *dev
device used to acquire the clock

struct clk *clk
clock source acquired with devm_clk_get()

Note

drivers must ensure that all clk_enable calls made on this clock source are balanced by clk_disable calls prior to calling this function.

Description

clk_put should not be called from within interrupt context.

long **clk_round_rate**(struct *clk* *clk, unsigned long rate)
    adjust a rate to the exact rate a clock can provide

Parameters

struct clk *clk
    clock source

unsigned long rate
    desired clock rate in Hz

Description

This answers the question "if I were to pass **rate** to *clk_set_rate()*, what clock rate would I end up with?" without changing the hardware in any way. In other words:

    rate = clk_round_rate(clk, r);

and:

    clk_set_rate(clk, r); rate = clk_get_rate(clk);

are equivalent except the former does not modify the clock hardware in any way.

Returns rounded clock rate in Hz, or negative errno.

int **clk_set_rate**(struct *clk* *clk, unsigned long rate)
    set the clock rate for a clock source

Parameters

struct clk *clk
    clock source

unsigned long rate
    desired clock rate in Hz

Description

Updating the rate starts at the top-most affected clock and then walks the tree down to the bottom-most clock that needs updating.

Returns success (0) or negative errno.

int **clk_set_rate_exclusive**(struct *clk* *clk, unsigned long rate)
    set the clock rate and claim exclusivity over clock source

Parameters

struct clk *clk
    clock source

unsigned long rate
    desired clock rate in Hz

Description

This helper function allows drivers to atomically set the rate of a producer and claim exclusivity over the rate control of the producer.

It is essentially a combination of *clk_set_rate()* and clk_rate_exclusite_get(). Caller must balance this call with a call to *clk_rate_exclusive_put()*

Returns success (0) or negative errno.

bool **clk_has_parent**(const struct *clk* *clk, const struct *clk* *parent)

check if a clock is a possible parent for another

Parameters

const struct clk *clk
clock source

const struct clk *parent
parent clock source

Description

This function can be used in drivers that need to check that a clock can be the parent of another without actually changing the parent.

Returns true if **parent** is a possible parent for **clk**, false otherwise.

int **clk_set_rate_range**(struct *clk* *clk, unsigned long min, unsigned long max)

set a rate range for a clock source

Parameters

struct clk *clk
clock source

unsigned long min
desired minimum clock rate in Hz, inclusive

unsigned long max
desired maximum clock rate in Hz, inclusive

Description

Returns success (0) or negative errno.

int **clk_set_min_rate**(struct *clk* *clk, unsigned long rate)

set a minimum clock rate for a clock source

Parameters

struct clk *clk
clock source

unsigned long rate
desired minimum clock rate in Hz, inclusive

Description

Returns success (0) or negative errno.

int **clk_set_max_rate**(struct *clk* *clk, unsigned long rate)
set a maximum clock rate for a clock source

Parameters

struct clk *clk
clock source

unsigned long rate
desired maximum clock rate in Hz, inclusive

Description

Returns success (0) or negative errno.

int **clk_set_parent**(struct *clk* *clk, struct *clk* *parent)
set the parent clock source for this clock

Parameters

struct clk *clk
clock source

struct clk *parent
parent clock source

Description

Returns success (0) or negative errno.

struct *clk* ***clk_get_parent**(struct *clk* *clk)
get the parent clock source for this clock

Parameters

struct clk *clk
clock source

Description

Returns struct clk corresponding to parent clock source, or valid IS_ERR() condition containing errno.

struct clk ***clk_get_sys**(const char *dev_id, const char *con_id)
get a clock based upon the device name

Parameters

const char *dev_id
device name

const char *con_id
connection ID

Description

Returns a struct clk corresponding to the clock producer, or valid IS_ERR() condition containing errno. The implementation uses **dev_id** and **con_id** to determine the clock consumer, and thereby the clock producer. In contrast to *clk_get()* this function takes the device name instead of the device itself for identification.

Drivers must assume that the clock source is not enabled.

clk_get_sys should not be called from within interrupt context.

int **clk_save_context**(void)

save clock context for poweroff

Parameters

void
no arguments

Description

Saves the context of the clock register for powerstates in which the contents of the registers will be lost. Occurs deep within the suspend code so locking is not necessary.

void **clk_restore_context**(void)

restore clock context after poweroff

Parameters

void
no arguments

Description

This occurs with all clocks enabled. Occurs deep within the resume code so locking is not necessary.

int **clk_drop_range**(struct *clk* *clk)

Reset any range set on that clock

Parameters

struct clk *clk
clock source

Description

Returns success (0) or negative errno.

struct clk ***clk_get_optional**(struct device *dev, const char *id)

lookup and obtain a reference to an optional clock producer.

Parameters

struct device *dev
device for clock "consumer"

const char *id
clock consumer ID

Description

Behaves the same as *clk_get()* except where there is no clock producer. In this case, instead of returning -ENOENT, the function returns NULL.

1.1.16 Synchronization Primitives

Read-Copy Update (RCU)

bool **same_state_synchronize_rcu**(unsigned long oldstate1, unsigned long oldstate2)

Are two old-state values identical?

Parameters

unsigned long oldstate1
First old-state value.

unsigned long oldstate2
Second old-state value.

Description

The two old-state values must have been obtained from either *get_state_synchronize_rcu()*, *start_poll_synchronize_rcu()*, or *get_completed_synchronize_rcu()*. Returns **true** if the two values are identical and **false** otherwise. This allows structures whose lifetimes are tracked by old-state values to push these values to a list header, allowing those structures to be slightly smaller.

bool **rcu_trace_implies_rcu_gp**(void)

does an RCU Tasks Trace grace period imply an RCU grace period?

Parameters

void
no arguments

Description

As an accident of implementation, an RCU Tasks Trace grace period also acts as an RCU grace period. However, this could change at any time. Code relying on this accident must call this function to verify that this accident is still happening.

You have been warned!

cond_resched_tasks_rcu_qs

`cond_resched_tasks_rcu_qs ()`

Report potential quiescent states to RCU

Parameters

Description

This macro resembles cond_resched(), except that it is defined to report potential quiescent states to RCU-tasks even if the cond_resched() machinery were to be shut off, as some advocate for PREEMPTION kernels.

rcu_softirq_qs_periodic

`rcu_softirq_qs_periodic (old_ts)`

Report RCU and RCU-Tasks quiescent states

Parameters

old_ts
jiffies at start of processing.

Description

This helper is for long-running softirq handlers, such as NAPI threads in networking. The caller should initialize the variable passed in as **old_ts** at the beginning of the softirq handler. When invoked frequently, this macro will invoke rcu_softirq_qs() every 100 milliseconds thereafter, which will provide both RCU and RCU-Tasks quiescent states. Note that this macro modifies its old_ts argument.

Because regions of code that have disabled softirq act as RCU read-side critical sections, this macro should be invoked with softirq (and preemption) enabled.

The macro is not needed when CONFIG_PREEMPT_RT is defined. RT kernels would have more chance to invoke schedule() calls and provide necessary quiescent states. As a contrast, calling cond_resched() only won't achieve the same effect because cond_resched() does not provide RCU-Tasks quiescent states.

RCU_LOCKDEP_WARN

`RCU_LOCKDEP_WARN (c, s)`

emit lockdep splat if specified condition is met

Parameters

c
condition to check

s
informative message

Description

This checks debug_lockdep_rcu_enabled() before checking (c) to prevent early boot splats due to lockdep not yet being initialized, and rechecks it after checking (c) to prevent false-positive splats due to races with lockdep being disabled. See commit 3066820034b5dd ("rcu: Reject *RCU_LOCKDEP_WARN()* false positives") for more detail.

unrcu_pointer

`unrcu_pointer (p)`

mark a pointer as not being RCU protected

Parameters

p
pointer needing to lose its __rcu property

Description

Converts **p** from an __rcu pointer to a __kernel pointer. This allows an __rcu pointer to be used with xchg() and friends.

RCU_INITIALIZER

`RCU_INITIALIZER (v)`

statically initialize an RCU-protected global variable

Parameters

v
: The value to statically initialize with.

rcu_assign_pointer

`rcu_assign_pointer (p, v)`

assign to RCU-protected pointer

Parameters

p
: pointer to assign to

v
: value to assign (publish)

Description

Assigns the specified value to the specified RCU-protected pointer, ensuring that any concurrent RCU readers will see any prior initialization.

Inserts memory barriers on architectures that require them (which is most of them), and also prevents the compiler from reordering the code that initializes the structure after the pointer assignment. More importantly, this call documents which pointers will be dereferenced by RCU read-side code.

In some special cases, you may use *RCU_INIT_POINTER()* instead of rcu_assign_pointer(). *RCU_INIT_POINTER()* is a bit faster due to the fact that it does not constrain either the CPU or the compiler. That said, using *RCU_INIT_POINTER()* when you should have used rcu_assign_pointer() is a very bad thing that results in impossible-to-diagnose memory corruption. So please be careful. See the *RCU_INIT_POINTER()* comment header for details.

Note that rcu_assign_pointer() evaluates each of its arguments only once, appearances notwithstanding. One of the "extra" evaluations is in typeof() and the other visible only to sparse (__CHECKER__), neither of which actually execute the argument. As with most cpp macros, this execute-arguments-only-once property is important, so please be careful when making changes to rcu_assign_pointer() and the other macros that it invokes.

rcu_replace_pointer

`rcu_replace_pointer (rcu_ptr, ptr, c)`

replace an RCU pointer, returning its old value

Parameters

rcu_ptr
: RCU pointer, whose old value is returned

ptr
: regular pointer

c
: the lockdep conditions under which the dereference will take place

Description

Perform a replacement, where **rcu_ptr** is an RCU-annotated pointer and **c** is the lockdep argument that is passed to the *rcu_dereference_protected()* call used to read that pointer. The old value of **rcu_ptr** is returned, and **rcu_ptr** is set to **ptr**.

rcu_access_pointer

`rcu_access_pointer (p)`

fetch RCU pointer with no dereferencing

Parameters

p
The pointer to read

Description

Return the value of the specified RCU-protected pointer, but omit the lockdep checks for being in an RCU read-side critical section. This is useful when the value of this pointer is accessed, but the pointer is not dereferenced, for example, when testing an RCU-protected pointer against NULL. Although *rcu_access_pointer()* may also be used in cases where update-side locks prevent the value of the pointer from changing, you should instead use *rcu_dereference_protected()* for this use case. Within an RCU read-side critical section, there is little reason to use *rcu_access_pointer()*.

It is usually best to test the *rcu_access_pointer()* return value directly in order to avoid accidental dereferences being introduced by later inattentive changes. In other words, assigning the *rcu_access_pointer()* return value to a local variable results in an accident waiting to happen.

It is also permissible to use *rcu_access_pointer()* when read-side access to the pointer was removed at least one grace period ago, as is the case in the context of the RCU callback that is freeing up the data, or after a synchronize_rcu() returns. This can be useful when tearing down multi-linked structures after a grace period has elapsed. However, *rcu_dereference_protected()* is normally preferred for this use case.

rcu_dereference_check

`rcu_dereference_check (p, c)`

rcu_dereference with debug checking

Parameters

p
The pointer to read, prior to dereferencing

c
The conditions under which the dereference will take place

Description

Do an rcu_dereference(), but check that the conditions under which the dereference will take place are correct. Typically the conditions indicate the various locking conditions that should be held at that point. The check should return true if the conditions are satisfied. An implicit check for being in an RCU read-side critical section (rcu_read_lock()) is included.

For example:

bar = rcu_dereference_check(foo->bar, lockdep_is_held(`foo->lock`));

could be used to indicate to lockdep that foo->bar may only be dereferenced if either rcu_read_lock() is held, or that the lock required to replace the bar struct at foo->bar is held.

Note that the list of conditions may also include indications of when a lock need not be held, for example during initialisation or destruction of the target struct:

bar = rcu_dereference_check(foo->bar, lockdep_is_held(foo->lock) ||
atomic_read(foo->usage) == 0);

Inserts memory barriers on architectures that require them (currently only the Alpha), prevents the compiler from refetching (and from merging fetches), and, more importantly, documents exactly which pointers are protected by RCU and checks that the pointer is annotated as __rcu.

rcu_dereference_bh_check

`rcu_dereference_bh_check (p, c)`

rcu_dereference_bh with debug checking

Parameters

p
The pointer to read, prior to dereferencing

c
The conditions under which the dereference will take place

Description

This is the RCU-bh counterpart to *rcu_dereference_check()*. However, please note that starting in v5.0 kernels, vanilla RCU grace periods wait for local_bh_disable() regions of code in addition to regions of code demarked by rcu_read_lock() and rcu_read_unlock(). This means that synchronize_rcu(), call_rcu, and friends all take not only rcu_read_lock() but also *rcu_read_lock_bh()* into account.

rcu_dereference_sched_check

`rcu_dereference_sched_check (p, c)`

rcu_dereference_sched with debug checking

Parameters

p
The pointer to read, prior to dereferencing

c
The conditions under which the dereference will take place

Description

This is the RCU-sched counterpart to *rcu_dereference_check()*. However, please note that starting in v5.0 kernels, vanilla RCU grace periods wait for preempt_disable() regions of code in addition to regions of code demarked by rcu_read_lock() and rcu_read_unlock(). This means that synchronize_rcu(), call_rcu, and friends all take not only rcu_read_lock() but also *rcu_read_lock_sched()* into account.

rcu_dereference_protected

`rcu_dereference_protected (p, c)`

fetch RCU pointer when updates prevented

Parameters

p
The pointer to read, prior to dereferencing

c
The conditions under which the dereference will take place

Description

Return the value of the specified RCU-protected pointer, but omit the READ_ONCE(). This is useful in cases where update-side locks prevent the value of the pointer from changing. Please note that this primitive does *not* prevent the compiler from repeating this reference or combining it with other references, so it should not be used without protection of appropriate locks.

This function is only for update-side use. Using this function when protected only by rcu_read_lock() will result in infrequent but very ugly failures.

rcu_dereference

`rcu_dereference (p)`

fetch RCU-protected pointer for dereferencing

Parameters

p
The pointer to read, prior to dereferencing

Description

This is a simple wrapper around *rcu_dereference_check()*.

rcu_dereference_bh

`rcu_dereference_bh (p)`

fetch an RCU-bh-protected pointer for dereferencing

Parameters

p
The pointer to read, prior to dereferencing

Description

Makes *rcu_dereference_check()* do the dirty work.

rcu_dereference_sched

`rcu_dereference_sched (p)`

fetch RCU-sched-protected pointer for dereferencing

Parameters

p
The pointer to read, prior to dereferencing

Description

Makes *rcu_dereference_check()* do the dirty work.

rcu_pointer_handoff

`rcu_pointer_handoff (p)`

Hand off a pointer from RCU to other mechanism

Parameters

p
The pointer to hand off

Description

This is simply an identity function, but it documents where a pointer is handed off from RCU to some other synchronization mechanism, for example, reference counting or locking. In C11, it would map to kill_dependency(). It could be used as follows:

```
rcu_read_lock();
p = rcu_dereference(gp);
long_lived = is_long_lived(p);
if (long_lived) {
        if (!atomic_inc_not_zero(p->refcnt))
                long_lived = false;
        else
                p = rcu_pointer_handoff(p);
}
rcu_read_unlock();
```

void **rcu_read_lock**(void)

mark the beginning of an RCU read-side critical section

Parameters

void
no arguments

Description

When synchronize_rcu() is invoked on one CPU while other CPUs are within RCU read-side critical sections, then the synchronize_rcu() is guaranteed to block until after all the other CPUs exit their critical sections. Similarly, if call_rcu() is invoked on one CPU while other CPUs are within RCU read-side critical sections, invocation of the corresponding RCU callback is deferred until after the all the other CPUs exit their critical sections.

In v5.0 and later kernels, synchronize_rcu() and call_rcu() also wait for regions of code with preemption disabled, including regions of code with interrupts or softirqs disabled. In pre-v5.0 kernels, which define synchronize_sched(), only code enclosed within rcu_read_lock() and rcu_read_unlock() are guaranteed to be waited for.

Note, however, that RCU callbacks are permitted to run concurrently with new RCU read-side critical sections. One way that this can happen is via the following sequence of events: (1) CPU 0 enters an RCU read-side critical section, (2) CPU 1 invokes call_rcu() to register an RCU callback, (3) CPU 0 exits the RCU read-side critical section, (4) CPU 2 enters a RCU read-side critical section, (5) the RCU callback is invoked. This is legal, because the RCU read-side critical section that was running concurrently with the call_rcu() (and which therefore might be referencing something that the corresponding RCU callback would free up) has completed before the corresponding RCU callback is invoked.

RCU read-side critical sections may be nested. Any deferred actions will be deferred until the outermost RCU read-side critical section completes.

You can avoid reading and understanding the next paragraph by following this rule: don't put anything in an rcu_read_lock() RCU read-side critical section that would block in a !PREEMPTION kernel. But if you want the full story, read on!

In non-preemptible RCU implementations (pure TREE_RCU and TINY_RCU), it is illegal to block while in an RCU read-side critical section. In preemptible RCU implementations (PREEMPT_RCU) in CONFIG_PREEMPTION kernel builds, RCU read-side critical sections may be preempted, but explicit blocking is illegal. Finally, in preemptible RCU implementations in real-time (with -rt patchset) kernel builds, RCU read-side critical sections may be preempted and they may also block, but only when acquiring spinlocks that are subject to priority inheritance.

void **rcu_read_unlock**(void)

marks the end of an RCU read-side critical section.

Parameters

void
no arguments

Description

In almost all situations, rcu_read_unlock() is immune from deadlock. In recent kernels that have consolidated synchronize_sched() and synchronize_rcu_bh() into synchronize_rcu(), this deadlock immunity also extends to the scheduler's runqueue and priority-inheritance spinlocks, courtesy of the quiescent-state deferral that is carried out when rcu_read_unlock() is invoked with interrupts disabled.

See rcu_read_lock() for more information.

void **rcu_read_lock_bh**(void)

mark the beginning of an RCU-bh critical section

Parameters

void
no arguments

Description

This is equivalent to rcu_read_lock(), but also disables softirqs. Note that anything else that disables softirqs can also serve as an RCU read-side critical section. However, please note that this equivalence applies only to v5.0 and later. Before v5.0, rcu_read_lock() and *rcu_read_lock_bh()* were unrelated.

Note that *rcu_read_lock_bh()* and the matching *rcu_read_unlock_bh()* must occur in the same context, for example, it is illegal to invoke *rcu_read_unlock_bh()* from one task if the matching *rcu_read_lock_bh()* was invoked from some other task.

void **rcu_read_unlock_bh**(void)

marks the end of a softirq-only RCU critical section

Parameters

void
no arguments

Description

See *rcu_read_lock_bh()* for more information.

void **rcu_read_lock_sched**(void)

mark the beginning of a RCU-sched critical section

Parameters

void
no arguments

Description

This is equivalent to rcu_read_lock(), but also disables preemption. Read-side critical sections can also be introduced by anything else that disables preemption, including local_irq_disable() and friends. However, please note that the equivalence to rcu_read_lock() applies only to v5.0 and later. Before v5.0, rcu_read_lock() and *rcu_read_lock_sched()* were unrelated.

Note that *rcu_read_lock_sched()* and the matching *rcu_read_unlock_sched()* must occur in the same context, for example, it is illegal to invoke *rcu_read_unlock_sched()* from process context if the matching *rcu_read_lock_sched()* was invoked from an NMI handler.

void **rcu_read_unlock_sched**(void)

marks the end of a RCU-classic critical section

Parameters

void
no arguments

Description

See *rcu_read_lock_sched()* for more information.

RCU_INIT_POINTER

`RCU_INIT_POINTER (p, v)`

initialize an RCU protected pointer

Parameters

p
The pointer to be initialized.

v
The value to initialized the pointer to.

Description

Initialize an RCU-protected pointer in special cases where readers do not need ordering constraints on the CPU or the compiler. These special cases are:

1. This use of *RCU_INIT_POINTER()* is NULLing out the pointer *or*
2. The caller has taken whatever steps are required to prevent RCU readers from concurrently accessing this pointer *or*
3. The referenced data structure has already been exposed to readers either at compile time or via rcu_assign_pointer() *and*

   a. You have not made *any* reader-visible changes to this structure since then *or*

b. It is OK for readers accessing this structure from its new location to see the old state of the structure. (For example, the changes were to statistical counters or to other state where exact synchronization is not required.)

Failure to follow these rules governing use of *RCU_INIT_POINTER()* will result in impossible-to-diagnose memory corruption. As in the structures will look OK in crash dumps, but any concurrent RCU readers might see pre-initialized values of the referenced data structure. So please be very careful how you use *RCU_INIT_POINTER()*!!!

If you are creating an RCU-protected linked structure that is accessed by a single external-to-structure RCU-protected pointer, then you may use *RCU_INIT_POINTER()* to initialize the internal RCU-protected pointers, but you must use rcu_assign_pointer() to initialize the external-to-structure pointer *after* you have completely initialized the reader-accessible portions of the linked structure.

Note that unlike rcu_assign_pointer(), *RCU_INIT_POINTER()* provides no ordering guarantees for either the CPU or the compiler.

RCU_POINTER_INITIALIZER

`RCU_POINTER_INITIALIZER (p, v)`

statically initialize an RCU protected pointer

Parameters

p
The pointer to be initialized.

v
The value to initialized the pointer to.

Description

GCC-style initialization for an RCU-protected pointer in a structure field.

kfree_rcu

`kfree_rcu (ptr, rhf)`

kfree an object after a grace period.

Parameters

ptr
pointer to kfree for double-argument invocations.

rhf
the name of the struct rcu_head within the type of **ptr**.

Description

Many rcu callbacks functions just call kfree() on the base structure. These functions are trivial, but their size adds up, and furthermore when they are used in a kernel module, that module must invoke the high-latency *rcu_barrier()* function at module-unload time.

The kfree_rcu() function handles this issue. Rather than encoding a function address in the embedded rcu_head structure, kfree_rcu() instead encodes the offset of the rcu_head structure within the base structure. Because the functions are not allowed in the low-order 4096 bytes of kernel virtual memory, offsets up to 4095 bytes can be accommodated. If the offset is larger than 4095 bytes, a compile-time error will be generated in kvfree_rcu_arg_2(). If this error is triggered, you can either fall back to use of call_rcu() or rearrange the structure to position the rcu_head structure into the first 4096 bytes.

The object to be freed can be allocated either by kmalloc() or kmem_cache_alloc().

Note that the allowable offset might decrease in the future.

The BUILD_BUG_ON check must not involve any function calls, hence the checks are done in macros here.

kfree_rcu_mightsleep

`kfree_rcu_mightsleep (ptr)`

kfree an object after a grace period.

Parameters

ptr
pointer to kfree for single-argument invocations.

Description

When it comes to head-less variant, only one argument is passed and that is just a pointer which has to be freed after a grace period. Therefore the semantic is

kfree_rcu_mightsleep(ptr);

where **ptr** is the pointer to be freed by kvfree().

Please note, head-less way of freeing is permitted to use from a context that has to follow might_sleep() annotation. Otherwise, please switch and embed the rcu_head structure within the type of **ptr**.

void **rcu_head_init**(struct rcu_head *rhp)

Initialize rcu_head for *rcu_head_after_call_rcu()*

Parameters

struct rcu_head *rhp
The rcu_head structure to initialize.

Description

If you intend to invoke *rcu_head_after_call_rcu()* to test whether a given rcu_head structure has already been passed to call_rcu(), then you must also invoke this *rcu_head_init()* function on it just after allocating that structure. Calls to this function must not race with calls to call_rcu(), *rcu_head_after_call_rcu()*, or callback invocation.

bool **rcu_head_after_call_rcu**(struct rcu_head *rhp, rcu_callback_t f)

Has this rcu_head been passed to call_rcu()?

Parameters

struct rcu_head *rhp
The rcu_head structure to test.

rcu_callback_t f
The function passed to call_rcu() along with **rhp**.

Description

Returns **true** if the **rhp** has been passed to call_rcu() with **func**, and **false** otherwise. Emits a warning in any other case, including the case where **rhp** has already been invoked after a grace period. Calls to this function must not race with callback invocation. One way to avoid such races is to enclose the call to *rcu_head_after_call_rcu()* in an RCU read-side critical section that includes a read-side fetch of the pointer to the structure containing **rhp**.

int **rcu_is_cpu_rrupt_from_idle**(void)

see if 'interrupted' from idle

Parameters

void

no arguments

Description

If the current CPU is idle and running at a first-level (not nested) interrupt, or directly, from idle, return true.

The caller must have at least disabled IRQs.

void **rcu_irq_exit_check_preempt**(void)

Validate that scheduling is possible

Parameters

void

no arguments

void **__rcu_irq_enter_check_tick**(void)

Enable scheduler tick on CPU if RCU needs it.

Parameters

void

no arguments

Description

The scheduler tick is not normally enabled when CPUs enter the kernel from nohz_full userspace execution. After all, nohz_full userspace execution is an RCU quiescent state and the time executing in the kernel is quite short. Except of course when it isn't. And it is not hard to cause a large system to spend tens of seconds or even minutes looping in the kernel, which can cause a number of problems, include RCU CPU stall warnings.

Therefore, if a nohz_full CPU fails to report a quiescent state in a timely manner, the RCU grace-period kthread sets that CPU's ->rcu_urgent_qs flag with the expectation that the next interrupt or exception will invoke this function, which will turn on the scheduler tick, which will enable RCU to detect that CPU's quiescent states, for example, due to cond_resched() calls in CONFIG_PREEMPT=n kernels. The tick will be disabled once a quiescent state is reported for this CPU.

Of course, in carefully tuned systems, there might never be an interrupt or exception. In that case, the RCU grace-period kthread will eventually cause one to happen. However, in less carefully controlled environments, this function allows RCU to get what it needs without creating otherwise useless interruptions.

notrace bool **rcu_is_watching**(void)

RCU read-side critical sections permitted on current CPU?

Parameters

void
    no arguments

Description

Return **true** if RCU is watching the running CPU and **false** otherwise. An **true** return means that this CPU can safely enter RCU read-side critical sections.

Although calls to *rcu_is_watching()* from most parts of the kernel will return **true**, there are important exceptions. For example, if the current CPU is deep within its idle loop, in kernel entry/exit code, or offline, *rcu_is_watching()* will return **false**.

Make notrace because it can be called by the internal functions of ftrace, and making this notrace removes unnecessary recursion calls.

void **call_rcu_hurry**(struct rcu_head *head, rcu_callback_t func)
    Queue RCU callback for invocation after grace period, and flush all lazy callbacks (including the new one) to the main ->cblist while doing so.

Parameters

struct rcu_head *head
    structure to be used for queueing the RCU updates.

rcu_callback_t func
    actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all pre-existing RCU read-side critical sections have completed.

Use this API instead of call_rcu() if you don't want the callback to be invoked after very long periods of time, which can happen on systems without memory pressure and on systems which are lightly loaded or mostly idle. This function will cause callbacks to be invoked sooner than later at the expense of extra power. Other than that, this function is identical to, and reuses call_rcu()'s logic. Refer to call_rcu() for more details about memory ordering and other functionality.

void **call_rcu**(struct rcu_head *head, rcu_callback_t func)
    Queue an RCU callback for invocation after a grace period. By default the callbacks are 'lazy' and are kept hidden from the main ->cblist to prevent starting of grace periods too soon. If you desire grace periods to start very soon, use *call_rcu_hurry()*.

Parameters

struct rcu_head *head
    structure to be used for queueing the RCU updates.

rcu_callback_t func
    actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all pre-existing RCU read-side critical sections have completed. However, the callback function might well execute concurrently with RCU read-side critical sections that started after call_rcu() was invoked.

RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), and may be nested. In addition, but only in v5.0 and later, regions of code across which interrupts, preemption, or softirqs have been disabled also serve as RCU read-side critical sections. This includes hardware interrupt handlers, softirq handlers, and NMI handlers.

Note that all CPUs must agree that the grace period extended beyond all pre-existing RCU read-side critical section. On systems with more than one CPU, this means that when "func()" is invoked, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU read-side critical section whose beginning preceded the call to call_rcu(). It also means that each CPU executing an RCU read-side critical section that continues beyond the start of "func()" must have executed a memory barrier after the call_rcu() but before the beginning of that RCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked call_rcu() and CPU B invoked the resulting RCU callback function "func()", then both CPU A and CPU B are guaranteed to execute a full memory barrier during the time interval between the call to call_rcu() and the invocation of "func()" -- even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

Implementation of these memory-ordering guarantees is described here: *A Tour Through TREE_RCU's Grace-Period Memory Ordering*.

struct **kvfree_rcu_bulk_data**

single block to store kvfree_rcu() pointers

Definition:

```
struct kvfree_rcu_bulk_data {
    struct list_head list;
    struct rcu_gp_oldstate gp_snap;
    unsigned long nr_records;
    void *records[];
};
```

Members

list

List node. All blocks are linked between each other

gp_snap

Snapshot of RCU state for objects placed to this bulk

nr_records

Number of active pointers in the array

records

Array of the kvfree_rcu() pointers

struct **kfree_rcu_cpu_work**

single batch of kfree_rcu() requests

Definition:

```
struct kfree_rcu_cpu_work {
    struct rcu_work rcu_work;
    struct rcu_head *head_free;
    struct rcu_gp_oldstate head_free_gp_snap;
```

```
    struct list_head bulk_head_free[FREE_N_CHANNELS];
    struct kfree_rcu_cpu *krcp;
};
```

Members

rcu_work
Let *queue_rcu_work()* invoke workqueue handler after grace period

head_free
List of kfree_rcu() objects waiting for a grace period

head_free_gp_snap
Grace-period snapshot to check for attempted premature frees.

bulk_head_free
Bulk-List of kvfree_rcu() objects waiting for a grace period

krcp
Pointer to **kfree_rcu_cpu** structure

struct **kfree_rcu_cpu**
batch up kfree_rcu() requests for RCU grace period

Definition:

```
struct kfree_rcu_cpu {
    struct rcu_head *head;
    unsigned long head_gp_snap;
    atomic_t head_count;
    struct list_head bulk_head[FREE_N_CHANNELS];
    atomic_t bulk_count[FREE_N_CHANNELS];
    struct kfree_rcu_cpu_work krw_arr[KFREE_N_BATCHES];
    raw_spinlock_t lock;
    struct delayed_work monitor_work;
    bool initialized;
    struct delayed_work page_cache_work;
    atomic_t backoff_page_cache_fill;
    atomic_t work_in_progress;
    struct hrtimer hrtimer;
    struct llist_head bkvcache;
    int nr_bkv_objs;
};
```

Members

head
List of kfree_rcu() objects not yet waiting for a grace period

head_gp_snap
Snapshot of RCU state for objects placed to "**head**"

head_count
Number of objects in rcu_head singular list

bulk_head
Bulk-List of kvfree_rcu() objects not yet waiting for a grace period

bulk_count
Number of objects in bulk-list

krw_arr
Array of batches of kfree_rcu() objects waiting for a grace period

lock
Synchronize access to this structure

monitor_work
Promote **head** to **head_free** after KFREE_DRAIN_JIFFIES

initialized
The **rcu_work** fields have been initialized

page_cache_work
A work to refill the cache when it is empty

backoff_page_cache_fill
Delay cache refills

work_in_progress
Indicates that page_cache_work is running

hrtimer
A hrtimer for scheduling a page_cache_work

bkvcache

A simple cache list that contains objects for reuse purpose. In order to save some per-cpu space the list is singular. Even though it is lockless an access has to be protected by the per-cpu lock.

nr_bkv_objs
number of allocated objects at **bkvcache**.

Description

This is a per-CPU structure. The reason that it is not included in the rcu_data structure is to permit this code to be extracted from the RCU files. Such extraction could allow further optimization of the interactions with the slab allocators.

void **synchronize_rcu**(void)

wait until a grace period has elapsed.

Parameters

void
no arguments

Description

Control will return to the caller some time after a full grace period has elapsed, in other words after all currently executing RCU read-side critical sections have completed. Note, however, that upon return from synchronize_rcu(), the caller might well be executing concurrently with new RCU read-side critical sections that began while synchronize_rcu() was waiting.

RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), and may be nested. In addition, but only in v5.0 and later, regions of code across which interrupts, preemption, or softirqs have been disabled also serve as RCU read-side critical sections. This includes hardware interrupt handlers, softirq handlers, and NMI handlers.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when synchronize_rcu() returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU read-side critical section whose beginning preceded the call to synchronize_rcu(). In addition, each CPU having an RCU read-side critical section that extends beyond the return from synchronize_rcu() is guaranteed to have executed a full memory barrier after the beginning of synchronize_rcu() and before the beginning of that RCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked synchronize_rcu(), which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of synchronize_rcu() -- even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

Implementation of these memory-ordering guarantees is described here: *A Tour Through TREE_RCU's Grace-Period Memory Ordering*.

void **get_completed_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp)

Return a full pre-completed polled state cookie

Parameters

struct rcu_gp_oldstate *rgosp
Place to put state cookie

Description

Stores into **rgosp** a value that will always be treated by functions like *poll_state_synchronize_rcu_full()* as a cookie whose grace period has already completed.

unsigned long **get_state_synchronize_rcu**(void)

Snapshot current RCU state

Parameters

void
no arguments

Description

Returns a cookie that is used by a later call to *cond_synchronize_rcu()* or *poll_state_synchronize_rcu()* to determine whether or not a full grace period has elapsed in the meantime.

void **get_state_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp)

Snapshot RCU state, both normal and expedited

Parameters

struct rcu_gp_oldstate *rgosp
location to place combined normal/expedited grace-period state

Description

Places the normal and expedited grace-period states in **rgosp**. This state value can be passed to a later call to *cond_synchronize_rcu_full()* or *poll_state_synchronize_rcu_full()* to determine whether or not a grace period (whether normal or expedited) has elapsed in the meantime. The rcu_gp_oldstate structure takes up twice the memory of an unsigned long, but is guaranteed to see all grace periods. In contrast, the combined state occupies less memory, but can sometimes fail to take grace periods into account.

This does not guarantee that the needed grace period will actually start.

unsigned long **start_poll_synchronize_rcu**(void)

Snapshot and start RCU grace period

Parameters

void

no arguments

Description

Returns a cookie that is used by a later call to *cond_synchronize_rcu()* or *poll_state_synchronize_rcu()* to determine whether or not a full grace period has elapsed in the meantime. If the needed grace period is not already slated to start, notifies RCU core of the need for that grace period.

Interrupts must be enabled for the case where it is necessary to awaken the grace-period kthread.

void **start_poll_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp)

Take a full snapshot and start RCU grace period

Parameters

struct rcu_gp_oldstate *rgosp

value from *get_state_synchronize_rcu_full()* or *start_poll_synchronize_rcu_full()*

Description

Places the normal and expedited grace-period states in ***rgos**. This state value can be passed to a later call to *cond_synchronize_rcu_full()* or *poll_state_synchronize_rcu_full()* to determine whether or not a grace period (whether normal or expedited) has elapsed in the meantime. If the needed grace period is not already slated to start, notifies RCU core of the need for that grace period.

Interrupts must be enabled for the case where it is necessary to awaken the grace-period kthread.

bool **poll_state_synchronize_rcu**(unsigned long oldstate)

Has the specified RCU grace period completed?

Parameters

unsigned long oldstate

value from *get_state_synchronize_rcu()* or *start_poll_synchronize_rcu()*

Description

If a full RCU grace period has elapsed since the earlier call from which **oldstate** was obtained, return **true**, otherwise return **false**. If **false** is returned, it is the caller's responsibility to invoke this function later on until it does return **true**. Alternatively, the caller can explicitly wait for a grace period, for example, by passing **oldstate** to either *cond_synchronize_rcu()* or *cond_synchronize_rcu_expedited()* on the one hand or by directly invoking either synchronize_rcu() or *synchronize_rcu_expedited()* on the other.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than a billion grace periods (and way more on a 64-bit system!). Those needing to keep old state values for very long time periods (many hours even on 32-bit systems) should check them occasionally and either refresh them or set a flag indicating that the grace period has completed. Alternatively, they can use *get_completed_synchronize_rcu()* to get a guaranteed-completed grace-period state.

In addition, because oldstate compresses the grace-period state for both normal and expedited grace periods into a single unsigned long, it can miss a grace period when synchronize_rcu() runs concurrently with *synchronize_rcu_expedited()*. If this is unacceptable, please instead use the _full() variant of these polling APIs.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **oldstate**, and that returned at the end of this function.

bool **poll_state_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp)

Has the specified RCU grace period completed?

Parameters

struct rcu_gp_oldstate *rgosp

value from *get_state_synchronize_rcu_full()* or *start_poll_synchronize_rcu_full()*

Description

If a full RCU grace period has elapsed since the earlier call from which *rgosp was obtained, return **true*, otherwise return **false**. If **false** is returned, it is the caller's responsibility to invoke this function later on until it does return **true**. Alternatively, the caller can explicitly wait for a grace period, for example, by passing **rgosp** to *cond_synchronize_rcu()* or by directly invoking synchronize_rcu().

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than a billion grace periods (and way more on a 64-bit system!). Those needing to keep rcu_gp_oldstate values for very long time periods (many hours even on 32-bit systems) should check them occasionally and either refresh them or set a flag indicating that the grace period has completed. Alternatively, they can use *get_completed_synchronize_rcu_full()* to get a guaranteed-completed grace-period state.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **rgosp**, and that returned at the end of this function. And this guarantee requires that the root rcu_node structure's ->gp_seq field be checked instead of that of the rcu_state structure. The problem is that the just-ending grace-period's callbacks can be invoked between the time that the root rcu_node structure's ->gp_seq field is updated and the time that the rcu_state structure's ->gp_seq field is updated. Therefore, if a single synchronize_rcu() is to cause a subsequent *poll_state_synchronize_rcu_full()* to return **true**, then the root rcu_node structure is the one that needs to be polled.

void **cond_synchronize_rcu**(unsigned long oldstate)

Conditionally wait for an RCU grace period

Parameters

unsigned long oldstate
value from *get_state_synchronize_rcu()*, *start_poll_synchronize_rcu()*, or *start_poll_synchronize_rcu_expedited()*

Description

If a full RCU grace period has elapsed since the earlier call to *get_state_synchronize_rcu()* or *start_poll_synchronize_rcu()*, just return. Otherwise, invoke synchronize_rcu() to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for a couple of additional grace periods should be just fine.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **oldstate** and that returned at the end of this function.

void **cond_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp)
Conditionally wait for an RCU grace period

Parameters

struct rcu_gp_oldstate *rgosp
value from *get_state_synchronize_rcu_full()*, *start_poll_synchronize_rcu_full()*, or *start_poll_synchronize_rcu_expedited_full()*

Description

If a full RCU grace period has elapsed since the call to *get_state_synchronize_rcu_full()*, *start_poll_synchronize_rcu_full()*, or *start_poll_synchronize_rcu_expedited_full()* from which **rgosp** was obtained, just return. Otherwise, invoke synchronize_rcu() to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for a couple of additional grace periods should be just fine.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **rgosp** and that returned at the end of this function.

void **rcu_barrier**(void)
Wait until all in-flight call_rcu() callbacks complete.

Parameters

void
no arguments

Description

Note that this primitive does not necessarily wait for an RCU grace period to complete. For example, if there are no RCU callbacks queued anywhere in the system, then *rcu_barrier()* is within its rights to return immediately, without waiting for anything, much less an RCU grace period.

void **synchronize_rcu_expedited**(void)

Brute-force RCU grace period

Parameters

void
no arguments

Description

Wait for an RCU grace period, but expedite it. The basic idea is to IPI all non-idle non-nohz online CPUs. The IPI handler checks whether the CPU is in an RCU critical section, and if so, it sets a flag that causes the outermost rcu_read_unlock() to report the quiescent state for RCU-preempt or asks the scheduler for help for RCU-sched. On the other hand, if the CPU is not in an RCU read-side critical section, the IPI handler reports the quiescent state immediately.

Although this is a great improvement over previous expedited implementations, it is still unfriendly to real-time workloads, so is thus not recommended for any sort of common-case code. In fact, if you are using *synchronize_rcu_expedited()* in a loop, please restructure your code to batch your updates, and then use a single synchronize_rcu() instead.

This has the same semantics as (but is more brutal than) synchronize_rcu().

unsigned long **start_poll_synchronize_rcu_expedited**(void)

Snapshot current RCU state and start expedited grace period

Parameters

void
no arguments

Description

Returns a cookie to pass to a call to *cond_synchronize_rcu()*, *cond_synchronize_rcu_expedited()*, or *poll_state_synchronize_rcu()*, allowing them to determine whether or not any sort of grace period has elapsed in the meantime. If the needed expedited grace period is not already slated to start, initiates that grace period.

void **start_poll_synchronize_rcu_expedited_full**(struct rcu_gp_oldstate *rgosp)

Take a full snapshot and start expedited grace period

Parameters

struct rcu_gp_oldstate *rgosp
Place to put snapshot of grace-period state

Description

Places the normal and expedited grace-period states in rgosp. This state value can be passed to a later call to *cond_synchronize_rcu_full()* or *poll_state_synchronize_rcu_full()* to determine whether or not a grace period (whether normal or expedited) has elapsed in the meantime. If the needed expedited grace period is not already slated to start, initiates that grace period.

void **cond_synchronize_rcu_expedited**(unsigned long oldstate)

Conditionally wait for an expedited RCU grace period

Parameters

unsigned long oldstate
value from *get_state_synchronize_rcu()*, *start_poll_synchronize_rcu()*, or *start_poll_synchronize_rcu_expedited()*

Description

If any type of full RCU grace period has elapsed since the earlier call to *get_state_synchronize_rcu()*, *start_poll_synchronize_rcu()*, or *start_poll_synchronize_rcu_expedited()*, just return. Otherwise, invoke *synchronize_rcu_expedited()* to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for a couple of additional grace periods should be just fine.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **oldstate** and that returned at the end of this function.

void **cond_synchronize_rcu_expedited_full**(struct rcu_gp_oldstate *rgosp)
Conditionally wait for an expedited RCU grace period

Parameters

struct rcu_gp_oldstate *rgosp
value from *get_state_synchronize_rcu_full()*, *start_poll_synchronize_rcu_full()*, or *start_poll_synchronize_rcu_expedited_full()*

Description

If a full RCU grace period has elapsed since the call to *get_state_synchronize_rcu_full()*, *start_poll_synchronize_rcu_full()*, or *start_poll_synchronize_rcu_expedited_full()* from which **rgosp** was obtained, just return. Otherwise, invoke *synchronize_rcu_expedited()* to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for a couple of additional grace periods should be just fine.

This function provides the same memory-ordering guarantees that would be provided by a synchronize_rcu() that was invoked at the call to the function that provided **rgosp** and that returned at the end of this function.

bool **rcu_read_lock_held_common**(bool *ret)
might we be in RCU-sched read-side critical section?

Parameters

bool *ret
Best guess answer if lockdep cannot be relied on

Description

Returns true if lockdep must be ignored, in which case `*ret` contains the best guess described below. Otherwise returns false, in which case `*ret` tells the caller nothing and the caller should instead consult lockdep.

If CONFIG_DEBUG_LOCK_ALLOC is selected, set `*ret` to nonzero iff in an RCU-sched read-side critical section. In absence of CONFIG_DEBUG_LOCK_ALLOC, this assumes we are in an RCU-sched read-side critical section unless it can prove otherwise. Note that disabling of preemption (including disabling irqs) counts as an RCU-sched read-side critical section. This is useful for debug checks in functions that required that they be called within an RCU-sched read-side critical section.

Check debug_lockdep_rcu_enabled() to prevent false positives during boot and while lockdep is disabled.

Note that if the CPU is in the idle loop from an RCU point of view (ie: that we are in the section between ct_idle_enter() and ct_idle_exit()) then *rcu_read_lock_held()* sets `*ret` to false even if the CPU did an rcu_read_lock(). The reason for this is that RCU ignores CPUs that are in such a section, considering these as in extended quiescent state, so such a CPU is effectively never in an RCU read-side critical section regardless of what RCU primitives it invokes. This state of affairs is required --- we need to keep an RCU-free window in idle where the CPU may possibly enter into low power mode. This way we can notice an extended quiescent state to other CPUs that started a grace period. Otherwise we would delay any grace period as long as we run in the idle task.

Similarly, we avoid claiming an RCU read lock held if the current CPU is offline.

void **rcu_async_hurry**(void)

Make future async RCU callbacks not lazy.

Parameters

void
no arguments

Description

After a call to this function, future calls to call_rcu() will be processed in a timely fashion.

void **rcu_async_relax**(void)

Make future async RCU callbacks lazy.

Parameters

void
no arguments

Description

After a call to this function, future calls to call_rcu() will be processed in a lazy fashion.

void **rcu_expedite_gp**(void)

Expedite future RCU grace periods

Parameters

void
no arguments

Description

After a call to this function, future calls to synchronize_rcu() and friends act as the corresponding *synchronize_rcu_expedited()* function had instead been called.

void **rcu_unexpedite_gp**(void)

Cancel prior *rcu_expedite_gp()* invocation

Parameters

void
no arguments

Description

Undo a prior call to *rcu_expedite_gp()*. If all prior calls to *rcu_expedite_gp()* are undone by a subsequent call to *rcu_unexpedite_gp()*, and if the rcu_expedited sysfs/boot parameter is not set, then all subsequent calls to synchronize_rcu() and friends will return to their normal non-expedited behavior.

int **rcu_read_lock_held**(void)
might we be in RCU read-side critical section?

Parameters

void
no arguments

Description

If CONFIG_DEBUG_LOCK_ALLOC is selected, returns nonzero iff in an RCU read-side critical section. In absence of CONFIG_DEBUG_LOCK_ALLOC, this assumes we are in an RCU read-side critical section unless it can prove otherwise. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Checks debug_lockdep_rcu_enabled() to prevent false positives during boot and while lockdep is disabled.

Note that rcu_read_lock() and the matching rcu_read_unlock() must occur in the same context, for example, it is illegal to invoke rcu_read_unlock() in process context if the matching rcu_read_lock() was invoked from within an irq handler.

Note that rcu_read_lock() is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

int **rcu_read_lock_bh_held**(void)
might we be in RCU-bh read-side critical section?

Parameters

void
no arguments

Description

Check for bottom half being disabled, which covers both the CONFIG_PROVE_RCU and not cases. Note that if someone uses *rcu_read_lock_bh()*, but then later enables BH, lockdep (if enabled) will show the situation. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Check debug_lockdep_rcu_enabled() to prevent false positives during boot.

Note that *rcu_read_lock_bh()* is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

void **wakeme_after_rcu**(struct rcu_head *head)
Callback function to awaken a task after grace period

Parameters

struct rcu_head *head
Pointer to rcu_head member within rcu_synchronize structure

Description

Awaken the corresponding task now that a grace period has elapsed.

void **init_rcu_head_on_stack**(struct rcu_head *head)
initialize on-stack rcu_head for debugobjects

Parameters

struct rcu_head *head
pointer to rcu_head structure to be initialized

Description

This function informs debugobjects of a new rcu_head structure that has been allocated as an auto variable on the stack. This function is not required for rcu_head structures that are statically defined or that are dynamically allocated on the heap. This function has no effect for !CONFIG_DEBUG_OBJECTS_RCU_HEAD kernel builds.

void **destroy_rcu_head_on_stack**(struct rcu_head *head)
destroy on-stack rcu_head for debugobjects

Parameters

struct rcu_head *head
pointer to rcu_head structure to be initialized

Description

This function informs debugobjects that an on-stack rcu_head structure is about to go out of scope. As with *init_rcu_head_on_stack()*, this function is not required for rcu_head structures that are statically defined or that are dynamically allocated on the heap. Also as with *init_rcu_head_on_stack()*, this function has no effect for !CONFIG_DEBUG_OBJECTS_RCU_HEAD kernel builds.

unsigned long **get_completed_synchronize_rcu**(void)
Return a pre-completed polled state cookie

Parameters

void
no arguments

Description

Returns a value that will always be treated by functions like *poll_state_synchronize_rcu()* as a cookie whose grace period has already completed.

int **srcu_read_lock_held**(const struct srcu_struct *ssp)
might we be in SRCU read-side critical section?

Parameters

const struct srcu_struct *ssp
The srcu_struct structure to check

Description

If CONFIG_DEBUG_LOCK_ALLOC is selected, returns nonzero iff in an SRCU read-side critical section. In absence of CONFIG_DEBUG_LOCK_ALLOC, this assumes we are in an SRCU read-side critical section unless it can prove otherwise.

Checks debug_lockdep_rcu_enabled() to prevent false positives during boot and while lockdep is disabled.

Note that SRCU is based on its own statemachine and it doesn't relies on normal RCU, it can be called from the CPU which is in the idle loop from an RCU point of view or offline.

srcu_dereference_check

`srcu_dereference_check (p, ssp, c)`

fetch SRCU-protected pointer for later dereferencing

Parameters

p
: the pointer to fetch and protect for later dereferencing

ssp
: pointer to the srcu_struct, which is used to check that we really are in an SRCU read-side critical section.

c
: condition to check for update-side use

Description

If PROVE_RCU is enabled, invoking this outside of an RCU read-side critical section will result in an RCU-lockdep splat, unless **c** evaluates to 1. The **c** argument will normally be a logical expression containing lockdep_is_held() calls.

srcu_dereference

`srcu_dereference (p, ssp)`

fetch SRCU-protected pointer for later dereferencing

Parameters

p
: the pointer to fetch and protect for later dereferencing

ssp
: pointer to the srcu_struct, which is used to check that we really are in an SRCU read-side critical section.

Description

Makes *rcu_dereference_check()* do the dirty work. If PROVE_RCU is enabled, invoking this outside of an RCU read-side critical section will result in an RCU-lockdep splat.

srcu_dereference_notrace

`srcu_dereference_notrace (p, ssp)`

no tracing and no lockdep calls from here

Parameters

p
the pointer to fetch and protect for later dereferencing

ssp
pointer to the srcu_struct, which is used to check that we really are in an SRCU read-side critical section.

int **srcu_read_lock**(struct srcu_struct *ssp)
register a new reader for an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to register the new reader.

Description

Enter an SRCU read-side critical section. Note that SRCU read-side critical sections may be nested. However, it is illegal to call anything that waits on an SRCU grace period for the same srcu_struct, whether directly or indirectly. Please note that one way to indirectly wait on an SRCU grace period is to acquire a mutex that is held elsewhere while calling synchronize_srcu() or *synchronize_srcu_expedited()*.

Note that *srcu_read_lock()* and the matching *srcu_read_unlock()* must occur in the same context, for example, it is illegal to invoke *srcu_read_unlock()* in an irq handler if the matching *srcu_read_lock()* was invoked in process context.

int **srcu_read_lock_nmisafe**(struct srcu_struct *ssp)
register a new reader for an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to register the new reader.

Description

Enter an SRCU read-side critical section, but in an NMI-safe manner. See *srcu_read_lock()* for more information.

int **srcu_down_read**(struct srcu_struct *ssp)
register a new reader for an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to register the new reader.

Description

Enter a semaphore-like SRCU read-side critical section. Note that SRCU read-side critical sections may be nested. However, it is illegal to call anything that waits on an SRCU grace period for the same srcu_struct, whether directly or indirectly. Please note that one way to indirectly wait on an SRCU grace period is to acquire a mutex that is held elsewhere while calling synchronize_srcu() or *synchronize_srcu_expedited()*. But if you want lockdep to help you keep this stuff straight, you should instead use *srcu_read_lock()*.

The semaphore-like nature of *srcu_down_read()* means that the matching *srcu_up_read()* can be invoked from some other context, for example, from some other task or from an irq handler. However, neither *srcu_down_read()* nor *srcu_up_read()* may be invoked from an NMI handler.

Calls to *srcu_down_read()* may be nested, similar to the manner in which calls to down_read() may be nested.

void **srcu_read_unlock**(struct srcu_struct *ssp, int idx)
unregister a old reader from an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to unregister the old reader.

int idx
return value from corresponding *srcu_read_lock()*.

Description

Exit an SRCU read-side critical section.

void **srcu_read_unlock_nmisafe**(struct srcu_struct *ssp, int idx)
unregister a old reader from an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to unregister the old reader.

int idx
return value from corresponding *srcu_read_lock()*.

Description

Exit an SRCU read-side critical section, but in an NMI-safe manner.

void **srcu_up_read**(struct srcu_struct *ssp, int idx)
unregister a old reader from an SRCU-protected structure.

Parameters

struct srcu_struct *ssp
srcu_struct in which to unregister the old reader.

int idx
return value from corresponding *srcu_read_lock()*.

Description

Exit an SRCU read-side critical section, but not necessarily from the same context as the matching *srcu_down_read()*.

void **smp_mb__after_srcu_read_unlock**(void)
ensure full ordering after srcu_read_unlock

Parameters

void
no arguments

Description

Converts the preceding srcu_read_unlock into a two-way memory barrier.

Call this after srcu_read_unlock, to guarantee that all memory operations that occur after smp_mb__after_srcu_read_unlock will appear to happen after the preceding srcu_read_unlock.

int **init_srcu_struct**(struct srcu_struct *ssp)

initialize a sleep-RCU structure

Parameters

struct srcu_struct *ssp
structure to initialize.

Description

Must invoke this on a given srcu_struct before passing that srcu_struct to any other function. Each srcu_struct represents a separate domain of SRCU protection.

bool **srcu_readers_active**(struct srcu_struct *ssp)

returns true if there are readers. and false otherwise

Parameters

struct srcu_struct *ssp
which srcu_struct to count active readers (holding srcu_read_lock).

Description

Note that this is not an atomic primitive, and can therefore suffer severe errors when invoked on an active srcu_struct. That said, it can be useful as an error check at cleanup time.

void **cleanup_srcu_struct**(struct srcu_struct *ssp)

deconstruct a sleep-RCU structure

Parameters

struct srcu_struct *ssp
structure to clean up.

Description

Must invoke this after you are finished using a given srcu_struct that was initialized via *init_srcu_struct()*, else you leak memory.

void **call_srcu**(struct srcu_struct *ssp, struct rcu_head *rhp, rcu_callback_t func)

Queue a callback for invocation after an SRCU grace period

Parameters

struct srcu_struct *ssp
srcu_struct in queue the callback

struct rcu_head *rhp
structure to be used for queueing the SRCU callback.

rcu_callback_t func
function to be invoked after the SRCU grace period

Description

The callback function will be invoked some time after a full SRCU grace period elapses, in other words after all pre-existing SRCU read-side critical sections have completed. However, the callback function might well execute concurrently with other SRCU read-side critical sections that started after *call_srcu()* was invoked. SRCU read-side critical sections are delimited by *srcu_read_lock()* and *srcu_read_unlock()*, and may be nested.

The callback will be invoked from process context, but must nevertheless be fast and must not block.

void **synchronize_srcu_expedited**(struct srcu_struct *ssp)

Brute-force SRCU grace period

Parameters

struct srcu_struct *ssp
srcu_struct with which to synchronize.

Description

Wait for an SRCU grace period to elapse, but be more aggressive about spinning rather than blocking when waiting.

Note that *synchronize_srcu_expedited()* has the same deadlock and memory-ordering properties as does synchronize_srcu().

void **synchronize_srcu**(struct srcu_struct *ssp)

wait for prior SRCU read-side critical-section completion

Parameters

struct srcu_struct *ssp
srcu_struct with which to synchronize.

Description

Wait for the count to drain to zero of both indexes. To avoid the possible starvation of synchronize_srcu(), it waits for the count of the index=((->srcu_idx & 1) ^ 1) to drain to zero at first, and then flip the srcu_idx and wait for the count of the other index.

Can block; must be called from process context.

Note that it is illegal to call synchronize_srcu() from the corresponding SRCU read-side critical section; doing so will result in deadlock. However, it is perfectly legal to call synchronize_srcu() on one srcu_struct from some other srcu_struct's read-side critical section, as long as the resulting graph of srcu_structs is acyclic.

There are memory-ordering constraints implied by synchronize_srcu(). On systems with more than one CPU, when synchronize_srcu() returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last corresponding SRCU read-side critical section whose beginning preceded the call to synchronize_srcu(). In addition, each CPU having an SRCU read-side critical section that extends beyond the return from synchronize_srcu() is guaranteed to have executed a full memory barrier after the beginning of synchronize_srcu() and before the beginning of that SRCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked synchronize_srcu(), which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of synchronize_srcu(). This guarantee applies even if CPU A and CPU B are the same CPU, but again only if the system has more than one CPU.

Of course, these memory-ordering guarantees apply only when synchronize_srcu(), *srcu_read_lock()*, and *srcu_read_unlock()* are passed the same srcu_struct structure.

Implementation of these memory-ordering guarantees is similar to that of synchronize_rcu().

If SRCU is likely idle, expedite the first request. This semantic was provided by Classic SRCU, and is relied upon by its users, so TREE SRCU must also provide it. Note that detecting idleness is heuristic and subject to both false positives and negatives.

unsigned long **get_state_synchronize_srcu**(struct srcu_struct *ssp)

Provide an end-of-grace-period cookie

Parameters

struct srcu_struct *ssp
srcu_struct to provide cookie for.

Description

This function returns a cookie that can be passed to *poll_state_synchronize_srcu()*, which will return true if a full grace period has elapsed in the meantime. It is the caller's responsibility to make sure that grace period happens, for example, by invoking *call_srcu()* after return from *get_state_synchronize_srcu()*.

unsigned long **start_poll_synchronize_srcu**(struct srcu_struct *ssp)

Provide cookie and start grace period

Parameters

struct srcu_struct *ssp
srcu_struct to provide cookie for.

Description

This function returns a cookie that can be passed to *poll_state_synchronize_srcu()*, which will return true if a full grace period has elapsed in the meantime. Unlike *get_state_synchronize_srcu()*, this function also ensures that any needed SRCU grace period will be started. This convenience does come at a cost in terms of CPU overhead.

bool **poll_state_synchronize_srcu**(struct srcu_struct *ssp, unsigned long cookie)

Has cookie's grace period ended?

Parameters

struct srcu_struct *ssp
srcu_struct to provide cookie for.

unsigned long cookie
Return value from *get_state_synchronize_srcu()* or *start_poll_synchronize_srcu()*.

Description

This function takes the cookie that was returned from either *get_state_synchronize_srcu()* or *start_poll_synchronize_srcu()*, and returns **true** if an SRCU grace period elapsed since the time that the cookie was created.

Because cookies are finite in size, wrapping/overflow is possible. This is more pronounced on 32-bit systems where cookies are 32 bits, where in theory wrapping could happen in about

14 hours assuming 25-microsecond expedited SRCU grace periods. However, a more likely overflow lower bound is on the order of 24 days in the case of one-millisecond SRCU grace periods. Of course, wrapping in a 64-bit system requires geologic timespans, as in more than seven million years even for expedited SRCU grace periods.

Wrapping/overflow is much more of an issue for CONFIG_SMP=n systems that also have CONFIG_PREEMPTION=n, which selects Tiny SRCU. This uses a 16-bit cookie, which rcutorture routinely wraps in a matter of a few minutes. If this proves to be a problem, this counter will be expanded to the same size as for Tree SRCU.

void **srcu_barrier**(struct srcu_struct *ssp)

Wait until all in-flight *call_srcu()* callbacks complete.

Parameters

struct srcu_struct *ssp
srcu_struct on which to wait for in-flight callbacks.

unsigned long **srcu_batches_completed**(struct srcu_struct *ssp)

return batches completed.

Parameters

struct srcu_struct *ssp
srcu_struct on which to report batch completion.

Description

Report the number of batches, correlated with, but not necessarily precisely the same as, the number of grace periods that have elapsed.

void **hlist_bl_del_rcu**(struct hlist_bl_node *n)

deletes entry from hash list without re-initialization

Parameters

struct hlist_bl_node *n
the element to delete from the hash list.

Note

hlist_bl_unhashed() on entry does not return true after this, the entry is in an undefined state. It is useful for RCU based lockfree traversal.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the hash list.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_bl_add_head_rcu()* or *hlist_bl_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as hlist_bl_for_each_entry().

void **hlist_bl_add_head_rcu**(struct hlist_bl_node *n, struct hlist_bl_head *h)

Parameters

struct hlist_bl_node *n
the element to add to the hash list.

struct hlist_bl_head *h
the list to add to.

Description

Adds the specified element to the specified hlist_bl, while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_bl_add_head_rcu()* or *hlist_bl_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_bl_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs. Regardless of the type of CPU, the list-traversal primitive must be guarded by rcu_read_lock().

hlist_bl_for_each_entry_rcu

`hlist_bl_for_each_entry_rcu (tpos, pos, head, member)`

iterate over rcu list of given type

Parameters

tpos
the type * to use as a loop cursor.

pos
the `struct hlist_bl_node` to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_bl_node within the struct.

list_tail_rcu

`list_tail_rcu (head)`

returns the prev pointer of the head of the list

Parameters

head
the head of the list

Note

This should only be used with the list header, and even then only if *list_del()* and similar primitives are not also used on the list header.

void **list_add_rcu**(struct list_head *new, struct list_head *head)

add a new entry to rcu-protected list

Parameters

struct list_head *new
new entry to be added

struct list_head *head
list head to add it after

Description

Insert a new entry after the specified head. This is good for implementing stacks.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as list_add_rcu() or list_del_rcu(), running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as list_for_each_entry_rcu().

void **list_add_tail_rcu**(struct list_head *new, struct list_head *head)

add a new entry to rcu-protected list

Parameters

struct list_head *new

new entry to be added

struct list_head *head

list head to add it before

Description

Insert a new entry before the specified head. This is useful for implementing queues.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *list_add_tail_rcu()* or list_del_rcu(), running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as list_for_each_entry_rcu().

void **list_del_rcu**(struct list_head *entry)

deletes entry from list without re-initialization

Parameters

struct list_head *entry

the element to delete from the list.

Note

list_empty() on entry does not return true after this, the entry is in an undefined state. It is useful for RCU based lockfree traversal.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the list.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as list_del_rcu() or list_add_rcu(), running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as list_for_each_entry_rcu().

Note that the caller is not permitted to immediately free the newly deleted entry. Instead, either synchronize_rcu() or call_rcu() must be used to defer freeing until an RCU grace period has elapsed.

void **hlist_del_init_rcu**(struct hlist_node *n)

deletes entry from hash list with re-initialization

Parameters

struct hlist_node *n
the element to delete from the hash list.

Note

list_unhashed() on the node return true after this. It is useful for RCU based read lockfree traversal if the writer side must know if the list entry is still hashed or already unhashed.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the hash list and we can only zero the pprev pointer so list_unhashed() will return true after this.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or *hlist_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_for_each_entry_rcu()*.

void **list_replace_rcu**(struct list_head *old, struct list_head *new)
replace old entry by new one

Parameters

struct list_head *old
the element to be replaced

struct list_head *new
the new element to insert

Description

The **old** entry will be replaced with the **new** entry atomically.

Note

old should not be empty.

void **__list_splice_init_rcu**(struct list_head *list, struct list_head *prev, struct list_head *next, void (*sync)(void))
join an RCU-protected list into an existing list.

Parameters

struct list_head *list
the RCU-protected list to splice

struct list_head *prev
points to the last element of the existing list

struct list_head *next
points to the first element of the existing list

void (*sync)(void)
synchronize_rcu, synchronize_rcu_expedited, …

Description

The list pointed to by **prev** and **next** can be RCU-read traversed concurrently with this function.

Note that this function blocks.

Important note: the caller must take whatever action is necessary to prevent any other updates to the existing list. In principle, it is possible to modify the list as soon as sync() begins execution. If this sort of thing becomes necessary, an alternative version based on call_rcu() could be created. But only if -really- needed -- there is no shortage of RCU API members.

void **list_splice_init_rcu**(struct list_head *list, struct list_head *head, void (*sync)(void))

splice an RCU-protected list into an existing list, designed for stacks.

Parameters

struct list_head *list
the RCU-protected list to splice

struct list_head *head
the place in the existing list to splice the first list into

void (*sync)(void)
synchronize_rcu, synchronize_rcu_expedited, ...

void **list_splice_tail_init_rcu**(struct list_head *list, struct list_head *head, void (*sync)(void))

splice an RCU-protected list into an existing list, designed for queues.

Parameters

struct list_head *list
the RCU-protected list to splice

struct list_head *head
the place in the existing list to splice the first list into

void (*sync)(void)
synchronize_rcu, synchronize_rcu_expedited, ...

list_entry_rcu

`list_entry_rcu (ptr, type, member)`

get the struct for this entry

Parameters

ptr
the `struct list_head` pointer.

type
the type of the struct this is embedded in.

member
the name of the list_head within the struct.

Description

This primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().

list_first_or_null_rcu

`list_first_or_null_rcu (ptr, type, member)`

get the first element from a list

Parameters

ptr
: the list head to take the element from.

type
: the type of the struct this is embedded in.

member
: the name of the list_head within the struct.

Description

Note that if the list is empty, it returns NULL.

This primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().

list_next_or_null_rcu

`list_next_or_null_rcu (head, ptr, type, member)`

get the first element from a list

Parameters

head
: the head for the list.

ptr
: the list head to take the next element from.

type
: the type of the struct this is embedded in.

member
: the name of the list_head within the struct.

Description

Note that if the ptr is at the end of the list, NULL is returned.

This primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().

list_for_each_entry_rcu

`list_for_each_entry_rcu (pos, head, member, cond...)`

iterate over rcu list of given type

Parameters

pos
: the type * to use as a loop cursor.

head
: the head for your list.

member
: the name of the list_head within the struct.

cond...
  optional lockdep expression if called from non-RCU protection.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu() as long as the traversal is guarded by rcu_read_lock().

list_for_each_entry_srcu

`list_for_each_entry_srcu (pos, head, member, cond)`

  iterate over rcu list of given type

Parameters

pos
  the type * to use as a loop cursor.

head
  the head for your list.

member
  the name of the list_head within the struct.

cond
  lockdep expression for the lock required to traverse the list.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu() as long as the traversal is guarded by *srcu_read_lock()*. The lockdep expression *srcu_read_lock_held()* can be passed as the cond argument from read side.

list_entry_lockless

`list_entry_lockless (ptr, type, member)`

  get the struct for this entry

Parameters

ptr
  the `struct list_head` pointer.

type
  the type of the struct this is embedded in.

member
  the name of the list_head within the struct.

Description

This primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu(), but requires some implicit RCU read-side guarding. One example is running within a special exception-time environment where preemption is disabled and where lockdep cannot be invoked. Another example is when items are added to the list, but never deleted.

list_for_each_entry_lockless

`list_for_each_entry_lockless (pos, head, member)`

  iterate over rcu list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_struct within the struct.

Description

This primitive may safely run concurrently with the _rcu list-mutation primitives such as list_add_rcu(), but requires some implicit RCU read-side guarding. One example is running within a special exception-time environment where preemption is disabled and where lockdep cannot be invoked. Another example is when items are added to the list, but never deleted.

list_for_each_entry_continue_rcu

`list_for_each_entry_continue_rcu (pos, head, member)`

continue iteration over list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_head within the struct.

Description

Continue to iterate over list of given type, continuing after the current position which must have been in the list when the RCU read lock was taken. This would typically require either that you obtained the node from a previous walk of the list in the same RCU read-side critical section, or that you held some sort of non-RCU reference (such as a reference count) to keep the node alive *and* in the list.

This iterator is similar to *list_for_each_entry_from_rcu()* except this starts after the given position and that one starts at the given position.

list_for_each_entry_from_rcu

`list_for_each_entry_from_rcu (pos, head, member)`

iterate over a list from current point

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the list_node within the struct.

Description

Iterate over the tail of a list starting from a given position, which must have been in the list when the RCU read lock was taken. This would typically require either that you obtained the node from a previous walk of the list in the same RCU read-side critical section, or that you held some sort of non-RCU reference (such as a reference count) to keep the node alive *and* in the list.

This iterator is similar to `list_for_each_entry_continue_rcu()` except this starts from the given position and that one starts from the position after the given position.

void **hlist_del_rcu**(struct hlist_node *n)

deletes entry from hash list without re-initialization

Parameters

struct hlist_node *n
the element to delete from the hash list.

Note

list_unhashed() on entry does not return true after this, the entry is in an undefined state. It is useful for RCU based lockfree traversal.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the hash list.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or `hlist_del_rcu()`, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as `hlist_for_each_entry()`.

void **hlist_replace_rcu**(struct hlist_node *old, struct hlist_node *new)

replace old entry by new one

Parameters

struct hlist_node *old
the element to be replaced

struct hlist_node *new
the new element to insert

Description

The **old** entry will be replaced with the **new** entry atomically.

void **hlists_swap_heads_rcu**(struct hlist_head *left, struct hlist_head *right)

swap the lists the hlist heads point to

Parameters

struct hlist_head *left
The hlist head on the left

struct hlist_head *right
The hlist head on the right

Description

The lists start out as [left][node1 ...] and
[**right**][node2 ...]

The lists end up as [left][node2 ...]
[**right**][node1 ...]

void **hlist_add_head_rcu**(struct hlist_node *n, struct hlist_head *h)

Parameters

struct hlist_node *n
the element to add to the hash list.

struct hlist_head *h
the list to add to.

Description

Adds the specified element to the specified hlist, while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or *hlist_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs. Regardless of the type of CPU, the list-traversal primitive must be guarded by rcu_read_lock().

void **hlist_add_tail_rcu**(struct hlist_node *n, struct hlist_head *h)

Parameters

struct hlist_node *n
the element to add to the hash list.

struct hlist_head *h
the list to add to.

Description

Adds the specified element to the specified hlist, while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or *hlist_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs. Regardless of the type of CPU, the list-traversal primitive must be guarded by rcu_read_lock().

void **hlist_add_before_rcu**(struct hlist_node *n, struct hlist_node *next)

Parameters

struct hlist_node *n
the new element to add to the hash list.

struct hlist_node *next
the existing element to add the new element before.

Description

Adds the specified element to the specified hlist before the specified node while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or *hlist_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs.

void **hlist_add_behind_rcu**(struct hlist_node *n, struct hlist_node *prev)

Parameters

struct hlist_node *n
the new element to add to the hash list.

struct hlist_node *prev
the existing element to add the new element after.

Description

Adds the specified element to the specified hlist after the specified node while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as hlist_add_head_rcu() or *hlist_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs.

hlist_for_each_entry_rcu

`hlist_for_each_entry_rcu (pos, head, member, cond...)`

iterate over rcu list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_node within the struct.

cond...
optional lockdep expression if called from non-RCU protection.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as hlist_add_head_rcu() as long as the traversal is guarded by rcu_read_lock().

hlist_for_each_entry_srcu

`hlist_for_each_entry_srcu (pos, head, member, cond)`

iterate over rcu list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_node within the struct.

cond
lockdep expression for the lock required to traverse the list.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as hlist_add_head_rcu() as long as the traversal is guarded by *srcu_read_lock()*. The lockdep expression *srcu_read_lock_held()* can be passed as the cond argument from read side.

hlist_for_each_entry_rcu_notrace

`hlist_for_each_entry_rcu_notrace (pos, head, member)`

iterate over rcu list of given type (for tracing)

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_node within the struct.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as hlist_add_head_rcu() as long as the traversal is guarded by rcu_read_lock().

This is the same as *hlist_for_each_entry_rcu()* except that it does not do any RCU debugging or tracing.

hlist_for_each_entry_rcu_bh

`hlist_for_each_entry_rcu_bh (pos, head, member)`

iterate over rcu list of given type

Parameters

pos
the type * to use as a loop cursor.

head
the head for your list.

member
the name of the hlist_node within the struct.

Description

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as hlist_add_head_rcu() as long as the traversal is guarded by rcu_read_lock().

hlist_for_each_entry_continue_rcu

`hlist_for_each_entry_continue_rcu (pos, member)`

iterate over a hlist continuing after current point

Parameters

pos
the type * to use as a loop cursor.

member
the name of the hlist_node within the struct.

hlist_for_each_entry_continue_rcu_bh

`hlist_for_each_entry_continue_rcu_bh (pos, member)`

iterate over a hlist continuing after current point

Parameters

pos
the type * to use as a loop cursor.

member
the name of the hlist_node within the struct.

hlist_for_each_entry_from_rcu

`hlist_for_each_entry_from_rcu (pos, member)`

iterate over a hlist continuing from current point

Parameters

pos
the type * to use as a loop cursor.

member
the name of the hlist_node within the struct.

void **hlist_nulls_del_init_rcu**(struct hlist_nulls_node *n)

deletes entry from hash list with re-initialization

Parameters

struct hlist_nulls_node *n
the element to delete from the hash list.

Note

hlist_nulls_unhashed() on the node return true after this. It is useful for RCU based read lock-free traversal if the writer side must know if the list entry is still hashed or already unhashed.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the hash list and we can only zero the pprev pointer so list_unhashed() will return true after this.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_nulls_add_head_rcu()* or *hlist_nulls_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_nulls_for_each_entry_rcu()*.

hlist_nulls_first_rcu

`hlist_nulls_first_rcu (head)`

returns the first element of the hash list.

Parameters

head
the head of the list.

hlist_nulls_next_rcu

`hlist_nulls_next_rcu (node)`

returns the element of the list after **node**.

Parameters

node
element of the list.

void **hlist_nulls_del_rcu**(struct hlist_nulls_node *n)

deletes entry from hash list without re-initialization

Parameters

struct hlist_nulls_node *n
the element to delete from the hash list.

Note

hlist_nulls_unhashed() on entry does not return true after this, the entry is in an undefined state. It is useful for RCU based lockfree traversal.

Description

In particular, it means that we can not poison the forward pointers that may still be used for walking the hash list.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_nulls_add_head_rcu()* or *hlist_nulls_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as hlist_nulls_for_each_entry().

void **hlist_nulls_add_head_rcu**(struct hlist_nulls_node *n, struct hlist_nulls_head *h)

Parameters

struct hlist_nulls_node *n
the element to add to the hash list.

struct hlist_nulls_head *h
the list to add to.

Description

Adds the specified element to the specified hlist_nulls, while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_nulls_add_head_rcu()* or *hlist_nulls_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_nulls_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs. Regardless of the type of CPU, the list-traversal primitive must be guarded by rcu_read_lock().

void **hlist_nulls_add_tail_rcu**(struct hlist_nulls_node *n, struct hlist_nulls_head *h)

Parameters

struct hlist_nulls_node *n
the element to add to the hash list.

struct hlist_nulls_head *h
the list to add to.

Description

Adds the specified element to the specified hlist_nulls, while permitting racing traversals.

The caller must take whatever precautions are necessary (such as holding appropriate locks) to avoid racing with another list-mutation primitive, such as *hlist_nulls_add_head_rcu()* or *hlist_nulls_del_rcu()*, running on this same list. However, it is perfectly legal to run concurrently with the _rcu list-traversal primitives, such as *hlist_nulls_for_each_entry_rcu()*, used to prevent memory-consistency problems on Alpha CPUs. Regardless of the type of CPU, the list-traversal primitive must be guarded by rcu_read_lock().

hlist_nulls_for_each_entry_rcu

`hlist_nulls_for_each_entry_rcu (tpos, pos, head, member)`

iterate over rcu list of given type

Parameters

tpos
the type * to use as a loop cursor.

pos
the `struct hlist_nulls_node` to use as a loop cursor.

head
the head of the list.

member
the name of the hlist_nulls_node within the struct.

Description

The barrier() is needed to make sure compiler doesn't cache first element [1], as this loop can be restarted [2] [1] Documentation/memory-barriers.txt around line 1533 [2] *Using RCU hlist_nulls to protect list and objects* around line 146

hlist_nulls_for_each_entry_safe

`hlist_nulls_for_each_entry_safe (tpos, pos, head, member)`

iterate over list of given type safe against removal of list entry

Parameters

tpos
the type * to use as a loop cursor.

pos
the `struct hlist_nulls_node` to use as a loop cursor.

head
the head of the list.

member
the name of the hlist_nulls_node within the struct.

bool **rcu_sync_is_idle**(struct rcu_sync *rsp)

Are readers permitted to use their fastpaths?

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to use for synchronization

Description

Returns true if readers are permitted to use their fastpaths. Must be invoked within some flavor of RCU read-side critical section.

void **rcu_sync_init**(struct rcu_sync *rsp)

Initialize an rcu_sync structure

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to be initialized

void **rcu_sync_enter_start**(struct rcu_sync *rsp)

Force readers onto slow path for multiple updates

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to use for synchronization

Description

Must be called after *rcu_sync_init()* and before first use.

Ensures *rcu_sync_is_idle()* returns false and rcu_sync_{enter,exit}() pairs turn into NO-OPs.

void **rcu_sync_func**(struct rcu_head *rhp)

Callback function managing reader access to fastpath

Parameters

struct rcu_head *rhp
Pointer to rcu_head in rcu_sync structure to use for synchronization

Description

This function is passed to call_rcu() function by *rcu_sync_enter()* and *rcu_sync_exit()*, so that it is invoked after a grace period following the that invocation of enter/exit.

If it is called by *rcu_sync_enter()* it signals that all the readers were switched onto slow path.

If it is called by *rcu_sync_exit()* it takes action based on events that have taken place in the meantime, so that closely spaced *rcu_sync_enter()* and *rcu_sync_exit()* pairs need not wait for a grace period.

If another *rcu_sync_enter()* is invoked before the grace period ended, reset state to allow the next *rcu_sync_exit()* to let the readers back onto their fastpaths (after a grace period). If both another *rcu_sync_enter()* and its matching *rcu_sync_exit()* are invoked before the grace period ended, re-invoke call_rcu() on behalf of that *rcu_sync_exit()*. Otherwise, set all state back to idle so that readers can again use their fastpaths.

void **rcu_sync_enter**(struct rcu_sync *rsp)

Force readers onto slowpath

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to use for synchronization

Description

This function is used by updaters who need readers to make use of a slowpath during the update. After this function returns, all subsequent calls to *rcu_sync_is_idle()* will return false, which tells readers to stay off their fastpaths. A later call to *rcu_sync_exit()* re-enables reader fastpaths.

When called in isolation, *rcu_sync_enter()* must wait for a grace period, however, closely spaced calls to *rcu_sync_enter()* can optimize away the grace-period wait via a state machine implemented by *rcu_sync_enter()*, *rcu_sync_exit()*, and *rcu_sync_func()*.

void **rcu_sync_exit**(struct rcu_sync *rsp)

Allow readers back onto fast path after grace period

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to use for synchronization

Description

This function is used by updaters who have completed, and can therefore now allow readers to make use of their fastpaths after a grace period has elapsed. After this grace period has completed, all subsequent calls to *rcu_sync_is_idle()* will return true, which tells readers that they can once again use their fastpaths.

void **rcu_sync_dtor**(struct rcu_sync *rsp)

Clean up an rcu_sync structure

Parameters

struct rcu_sync *rsp
Pointer to rcu_sync structure to be cleaned up

struct **rcu_tasks_percpu**

Per-CPU component of definition for a Tasks-RCU-like mechanism.

Definition:

```
struct rcu_tasks_percpu {
    struct rcu_segcblist cblist;
    raw_spinlock_t __private lock;
    unsigned long rtp_jiffies;
    unsigned long rtp_n_lock_retries;
    struct timer_list lazy_timer;
    unsigned int urgent_gp;
    struct work_struct rtp_work;
    struct irq_work rtp_irq_work;
    struct rcu_head barrier_q_head;
    struct list_head rtp_blkd_tasks;
    int cpu;
    struct rcu_tasks *rtpp;
};
```

Members

cblist
Callback list.

lock
Lock protecting per-CPU callback list.

rtp_jiffies
Jiffies counter value for statistics.

rtp_n_lock_retries
Rough lock-contention statistic.

lazy_timer
Timer to unlazify callbacks.

urgent_gp
Number of additional non-lazy grace periods.

rtp_work
Work queue for invoking callbacks.

rtp_irq_work
IRQ work queue for deferred wakeups.

barrier_q_head
RCU callback for barrier operation.

rtp_blkd_tasks
List of tasks blocked as readers.

cpu
CPU number corresponding to this entry.

rtpp
Pointer to the rcu_tasks structure.

struct **rcu_tasks**
Definition for a Tasks-RCU-like mechanism.

Definition:

```
struct rcu_tasks {
    struct rcuwait cbs_wait;
    raw_spinlock_t cbs_gbl_lock;
    struct mutex tasks_gp_mutex;
    int gp_state;
    int gp_sleep;
    int init_fract;
    unsigned long gp_jiffies;
    unsigned long gp_start;
    unsigned long tasks_gp_seq;
    unsigned long n_ipis;
    unsigned long n_ipis_fails;
    struct task_struct *kthread_ptr;
    unsigned long lazy_jiffies;
    rcu_tasks_gp_func_t gp_func;
    pregp_func_t pregp_func;
    pertask_func_t pertask_func;
    postscan_func_t postscan_func;
    holdouts_func_t holdouts_func;
    postgp_func_t postgp_func;
    call_rcu_func_t call_func;
    struct rcu_tasks_percpu __percpu *rtpcpu;
    int percpu_enqueue_shift;
    int percpu_enqueue_lim;
    int percpu_dequeue_lim;
    unsigned long percpu_dequeue_gpseq;
    struct mutex barrier_q_mutex;
    atomic_t barrier_q_count;
    struct completion barrier_q_completion;
    unsigned long barrier_q_seq;
    char *name;
    char *kname;
};
```

Members

cbs_wait
RCU wait allowing a new callback to get kthread's attention.

cbs_gbl_lock
Lock protecting callback list.

tasks_gp_mutex
Mutex protecting grace period, needed during mid-boot dead zone.

gp_state
Grace period's most recent state transition (debugging).

gp_sleep

Per-grace-period sleep to prevent CPU-bound looping.

init_fract
Initial backoff sleep interval.

gp_jiffies
Time of last **gp_state** transition.

gp_start
Most recent grace-period start in jiffies.

tasks_gp_seq
Number of grace periods completed since boot.

n_ipis
Number of IPIs sent to encourage grace periods to end.

n_ipis_fails
Number of IPI-send failures.

kthread_ptr
This flavor's grace-period/callback-invocation kthread.

lazy_jiffies
Number of jiffies to allow callbacks to be lazy.

gp_func
This flavor's grace-period-wait function.

pregp_func
This flavor's pre-grace-period function (optional).

pertask_func
This flavor's per-task scan function (optional).

postscan_func
This flavor's post-task scan function (optional).

holdouts_func
This flavor's holdout-list scan function (optional).

postgp_func
This flavor's post-grace-period function (optional).

call_func
This flavor's call_rcu()-equivalent function.

rtpcpu
This flavor's rcu_tasks_percpu structure.

percpu_enqueue_shift
Shift down CPU ID this much when enqueuing callbacks.

percpu_enqueue_lim
Number of per-CPU callback queues in use for enqueuing.

percpu_dequeue_lim
Number of per-CPU callback queues in use for dequeuing.

percpu_dequeue_gpseq
RCU grace-period number to propagate enqueue limit to dequeuers.

barrier_q_mutex
Serialize barrier operations.

barrier_q_count
Number of queues being waited on.

barrier_q_completion
Barrier wait/wakeup mechanism.

barrier_q_seq
Sequence number for barrier operations.

name
This flavor's textual name.

kname
This flavor's kthread name.

void **call_rcu_tasks**(struct rcu_head *rhp, rcu_callback_t func)

Queue an RCU for invocation task-based grace period

Parameters

struct rcu_head *rhp
structure to be used for queueing the RCU updates.

rcu_callback_t func
actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all currently executing RCU read-side critical sections have completed. *call_rcu_tasks()* assumes that the read-side critical sections end at a voluntary context switch (not a preemption!), *cond_resched_tasks_rcu_qs()*, entry into idle, or transition to usermode execution. As such, there are no read-side primitives analogous to rcu_read_lock() and rcu_read_unlock() because this primitive is intended to determine that all tasks have passed through a safe state, not so much for data-structure synchronization.

See the description of call_rcu() for more detailed information on memory ordering guarantees.

void **synchronize_rcu_tasks**(void)

wait until an rcu-tasks grace period has elapsed.

Parameters

void
no arguments

Description

Control will return to the caller some time after a full rcu-tasks grace period has elapsed, in other words after all currently executing rcu-tasks read-side critical sections have elapsed. These read-side critical sections are delimited by calls to schedule(), *cond_resched_tasks_rcu_qs()*, idle execution, userspace execution, calls to *synchronize_rcu_tasks()*, and (in theory, anyway) cond_resched().

This is a very specialized primitive, intended only for a few uses in tracing and other situations requiring manipulation of function preambles and profiling hooks. The *synchronize_rcu_tasks()* function is not (yet) intended for heavy use from multiple CPUs.

See the description of synchronize_rcu() for more detailed information on memory ordering guarantees.

void **rcu_barrier_tasks**(void)

Wait for in-flight *call_rcu_tasks()* callbacks.

Parameters

void
no arguments

Description

Although the current implementation is guaranteed to wait, it is not obligated to, for example, if there are no pending callbacks.

void **call_rcu_tasks_rude**(struct rcu_head *rhp, rcu_callback_t func)

Queue a callback rude task-based grace period

Parameters

struct rcu_head *rhp
structure to be used for queueing the RCU updates.

rcu_callback_t func
actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all currently executing RCU read-side critical sections have completed. *call_rcu_tasks_rude()* assumes that the read-side critical sections end at context switch, *cond_resched_tasks_rcu_qs()*, or transition to usermode execution (as usermode execution is schedulable). As such, there are no read-side primitives analogous to rcu_read_lock() and rcu_read_unlock() because this primitive is intended to determine that all tasks have passed through a safe state, not so much for data-structure synchronization.

See the description of call_rcu() for more detailed information on memory ordering guarantees.

void **synchronize_rcu_tasks_rude**(void)

wait for a rude rcu-tasks grace period

Parameters

void
no arguments

Description

Control will return to the caller some time after a rude rcu-tasks grace period has elapsed, in other words after all currently executing rcu-tasks read-side critical sections have elapsed. These read-side critical sections are delimited by calls to schedule(), *cond_resched_tasks_rcu_qs()*, userspace execution (which is a schedulable context), and (in theory, anyway) cond_resched().

This is a very specialized primitive, intended only for a few uses in tracing and other situations requiring manipulation of function preambles and profiling hooks. The *synchronize_rcu_tasks_rude()* function is not (yet) intended for heavy use from multiple CPUs.

See the description of synchronize_rcu() for more detailed information on memory ordering guarantees.

void **rcu_barrier_tasks_rude**(void)

Wait for in-flight *call_rcu_tasks_rude()* callbacks.

Parameters

void
no arguments

Description

Although the current implementation is guaranteed to wait, it is not obligated to, for example, if there are no pending callbacks.

void **call_rcu_tasks_trace**(struct rcu_head *rhp, rcu_callback_t func)

Queue a callback trace task-based grace period

Parameters

struct rcu_head *rhp
structure to be used for queueing the RCU updates.

rcu_callback_t func
actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a trace rcu-tasks grace period elapses, in other words after all currently executing trace rcu-tasks read-side critical sections have completed. These read-side critical sections are delimited by calls to *rcu_read_lock_trace()* and *rcu_read_unlock_trace()*.

See the description of call_rcu() for more detailed information on memory ordering guarantees.

void **synchronize_rcu_tasks_trace**(void)

wait for a trace rcu-tasks grace period

Parameters

void
no arguments

Description

Control will return to the caller some time after a trace rcu-tasks grace period has elapsed, in other words after all currently executing trace rcu-tasks read-side critical sections have elapsed. These read-side critical sections are delimited by calls to *rcu_read_lock_trace()* and *rcu_read_unlock_trace()*.

This is a very specialized primitive, intended only for a few uses in tracing and other situations requiring manipulation of function preambles and profiling hooks. The *synchronize_rcu_tasks_trace()* function is not (yet) intended for heavy use from multiple CPUs.

See the description of synchronize_rcu() for more detailed information on memory ordering guarantees.

void **rcu_barrier_tasks_trace**(void)

Wait for in-flight *call_rcu_tasks_trace()* callbacks.

Parameters

void
no arguments

Description

Although the current implementation is guaranteed to wait, it is not obligated to, for example, if there are no pending callbacks.

bool **rcu_gp_might_be_stalled**(void)

Is it likely that the grace period is stalled?

Parameters

void
no arguments

Description

Returns **true** if the current grace period is sufficiently old that it is reasonable to assume that it might be stalled. This can be useful when deciding whether to allocate memory to enable RCU-mediated freeing on the one hand or just invoking synchronize_rcu() on the other. The latter is preferable when the grace period is stalled.

Note that sampling of the .gp_start and .gp_seq fields must be done carefully to avoid false positives at the beginnings and ends of grace periods.

void **rcu_cpu_stall_reset**(void)

restart stall-warning timeout for current grace period

Parameters

void
no arguments

Description

To perform the reset request from the caller, disable stall detection until 3 fqs loops have passed. This is required to ensure a fresh jiffies is loaded. It should be safe to do from the fqs loop as enough timer interrupts and context switches should have passed.

The caller must disable hard irqs.

void **rcu_read_lock_trace**(void)

mark beginning of RCU-trace read-side critical section

Parameters

void
no arguments

Description

When *synchronize_rcu_tasks_trace()* is invoked by one task, then that task is guaranteed to block until all other tasks exit their read-side critical sections. Similarly, if call_rcu_trace() is invoked on one task while other tasks are within RCU read-side critical sections, invocation of the corresponding RCU callback is deferred until after the all the other tasks exit their critical sections.

For more details, please see the documentation for rcu_read_lock().

void **rcu_read_unlock_trace**(void)

mark end of RCU-trace read-side critical section

Parameters

void
no arguments

Description

Pairs with a preceding call to *rcu_read_lock_trace()*, and nesting is allowed. Invoking a *rcu_read_unlock_trace()* when there is no matching *rcu_read_lock_trace()* is verboten, and will result in lockdep complaints.

For more details, please see the documentation for rcu_read_unlock().

synchronize_rcu_mult

`synchronize_rcu_mult (...)`

Wait concurrently for multiple grace periods

Parameters

...
List of call_rcu() functions for different grace periods to wait on

Description

This macro waits concurrently for multiple types of RCU grace periods. For example, synchronize_rcu_mult(call_rcu, call_rcu_tasks) would wait on concurrent RCU and RCU-tasks grace periods. Waiting on a given SRCU domain requires you to write a wrapper function for that SRCU domain's *call_srcu()* function, with this wrapper supplying the pointer to the corresponding srcu_struct.

Note that *call_rcu_hurry()* should be used instead of call_rcu() because in kernels built with CONFIG_RCU_LAZY=y the delay between the invocation of call_rcu() and that of the corresponding RCU callback can be multiple seconds.

The first argument tells Tiny RCU's _wait_rcu_gp() not to bother waiting for RCU. The reason for this is because anywhere *synchronize_rcu_mult()* can be called is automatically already a full grace period.

void **rcuref_init**(rcuref_t *ref, unsigned int cnt)

Initialize a rcuref reference count with the given reference count

Parameters

rcuref_t *ref
Pointer to the reference count

unsigned int cnt
The initial reference count typically '1'

unsigned int **rcuref_read**(rcuref_t *ref)

Read the number of held reference counts of a rcuref

Parameters

rcuref_t *ref
Pointer to the reference count

Return

The number of held references (0 … N)

bool **rcuref_get**(rcuref_t *ref)

Acquire one reference on a rcuref reference count

Parameters

rcuref_t *ref
Pointer to the reference count

Description

Similar to atomic_inc_not_zero() but saturates at RCUREF_MAXREF.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See documentation in lib/rcuref.c

Return

False if the attempt to acquire a reference failed. This happens when the last reference has been put already

True if a reference was successfully acquired

bool **rcuref_put_rcusafe**(rcuref_t *ref)

- Release one reference for a rcuref reference count RCU safe

Parameters

rcuref_t *ref
Pointer to the reference count

Description

Provides release memory ordering, such that prior loads and stores are done before, and provides an acquire ordering on success such that free() must come after.

Can be invoked from contexts, which guarantee that no grace period can happen which would free the object concurrently if the decrement drops the last reference and the slowpath races against a concurrent get() and put() pair. rcu_read_lock()'ed and atomic contexts qualify.

Return

True if this was the last reference with no future references possible. This signals the caller that it can safely release the object which is protected by the reference counter.

False if there are still active references or the put() raced with a concurrent get()/put() pair. Caller is not allowed to release the protected object.

bool **rcuref_put**(rcuref_t *ref)

- Release one reference for a rcuref reference count

Parameters

rcuref_t *ref
Pointer to the reference count

Description

Can be invoked from any context.

Provides release memory ordering, such that prior loads and stores are done before, and provides an acquire ordering on success such that free() must come after.

Return

True if this was the last reference with no future references possible. This signals the caller that it can safely schedule the object, which is protected by the reference counter, for deconstruction.

False if there are still active references or the put() raced with a concurrent get()/put() pair. Caller is not allowed to deconstruct the protected object.

bool **same_state_synchronize_rcu_full**(struct rcu_gp_oldstate *rgosp1, struct rcu_gp_oldstate *rgosp2)

Are two old-state values identical?

Parameters

struct rcu_gp_oldstate *rgosp1
First old-state value.

struct rcu_gp_oldstate *rgosp2
Second old-state value.

Description

The two old-state values must have been obtained from either *get_state_synchronize_rcu_full()*, *start_poll_synchronize_rcu_full()*, or *get_completed_synchronize_rcu_full()*. Returns **true** if the two values are identical and **false** otherwise. This allows structures whose lifetimes are tracked by old-state values to push these values to a list header, allowing those structures to be slightly smaller.

Note that equality is judged on a bitwise basis, so that an **rcu_gp_oldstate** structure with an already-completed state in one field will compare not-equal to a structure with an already-completed state in the other field. After all, the **rcu_gp_oldstate** structure is opaque so how did such a situation come to pass in the first place?

1.2 Workqueue

Date
September, 2010

Author
Tejun Heo <tj@kernel.org>

Author
Florian Mickler <florian@mickler.org>

1.2.1 Introduction

There are many cases where an asynchronous process execution context is needed and the workqueue (wq) API is the most commonly used mechanism for such cases.

When such an asynchronous execution context is needed, a work item describing which function to execute is put on a queue. An independent thread serves as the asynchronous execution context. The queue is called workqueue and the thread is called worker.

While there are work items on the workqueue the worker executes the functions associated with the work items one after the other. When there is no work item left on the workqueue the worker becomes idle. When a new work item gets queued, the worker begins executing again.

1.2.2 Why Concurrency Managed Workqueue?

In the original wq implementation, a multi threaded (MT) wq had one worker thread per CPU and a single threaded (ST) wq had one worker thread system-wide. A single MT wq needed to keep around the same number of workers as the number of CPUs. The kernel grew a lot of MT wq users over the years and with the number of CPU cores continuously rising, some systems saturated the default 32k PID space just booting up.

Although MT wq wasted a lot of resource, the level of concurrency provided was unsatisfactory. The limitation was common to both ST and MT wq albeit less severe on MT. Each wq maintained its own separate worker pool. An MT wq could provide only one execution context per CPU while an ST wq one for the whole system. Work items had to compete for those very limited execution contexts leading to various problems including proneness to deadlocks around the single execution context.

The tension between the provided level of concurrency and resource usage also forced its users to make unnecessary tradeoffs like libata choosing to use ST wq for polling PIOs and accepting an unnecessary limitation that no two polling PIOs can progress at the same time. As MT wq don't provide much better concurrency, users which require higher level of concurrency, like async or fscache, had to implement their own thread pool.

Concurrency Managed Workqueue (cmwq) is a reimplementation of wq with focus on the following goals.

- Maintain compatibility with the original workqueue API.
- Use per-CPU unified worker pools shared by all wq to provide flexible level of concurrency on demand without wasting a lot of resource.

- Automatically regulate worker pool and level of concurrency so that the API users don't need to worry about such details.

1.2.3 The Design

In order to ease the asynchronous execution of functions a new abstraction, the work item, is introduced.

A work item is a simple struct that holds a pointer to the function that is to be executed asynchronously. Whenever a driver or subsystem wants a function to be executed asynchronously it has to set up a work item pointing to that function and queue that work item on a workqueue.

Special purpose threads, called worker threads, execute the functions off of the queue, one after the other. If no work is queued, the worker threads become idle. These worker threads are managed in so called worker-pools.

The cmwq design differentiates between the user-facing workqueues that subsystems and drivers queue work items on and the backend mechanism which manages worker-pools and processes the queued work items.

There are two worker-pools, one for normal work items and the other for high priority ones, for each possible CPU and some extra worker-pools to serve work items queued on unbound workqueues - the number of these backing pools is dynamic.

Subsystems and drivers can create and queue work items through special workqueue API functions as they see fit. They can influence some aspects of the way the work items are executed by setting flags on the workqueue they are putting the work item on. These flags include things like CPU locality, concurrency limits, priority and more. To get a detailed overview refer to the API description of `alloc_workqueue()` below.

When a work item is queued to a workqueue, the target worker-pool is determined according to the queue parameters and workqueue attributes and appended on the shared worklist of the worker-pool. For example, unless specifically overridden, a work item of a bound workqueue will be queued on the worklist of either normal or highpri worker-pool that is associated to the CPU the issuer is running on.

For any worker pool implementation, managing the concurrency level (how many execution contexts are active) is an important issue. cmwq tries to keep the concurrency at a minimal but sufficient level. Minimal to save resources and sufficient in that the system is used at its full capacity.

Each worker-pool bound to an actual CPU implements concurrency management by hooking into the scheduler. The worker-pool is notified whenever an active worker wakes up or sleeps and keeps track of the number of the currently runnable workers. Generally, work items are not expected to hog a CPU and consume many cycles. That means maintaining just enough concurrency to prevent work processing from stalling should be optimal. As long as there are one or more runnable workers on the CPU, the worker-pool doesn't start execution of a new work, but, when the last running worker goes to sleep, it immediately schedules a new worker so that the CPU doesn't sit idle while there are pending work items. This allows using a minimal number of workers without losing execution bandwidth.

Keeping idle workers around doesn't cost other than the memory space for kthreads, so cmwq holds onto idle ones for a while before killing them.

For unbound workqueues, the number of backing pools is dynamic. Unbound workqueue can be assigned custom attributes using `apply_workqueue_attrs()` and workqueue will automatically create backing worker pools matching the attributes. The responsibility of regulating concurrency level is on the users. There is also a flag to mark a bound wq to ignore the concurrency management. Please refer to the API section for details.

Forward progress guarantee relies on that workers can be created when more execution contexts are necessary, which in turn is guaranteed through the use of rescue workers. All work items which might be used on code paths that handle memory reclaim are required to be queued on wq's that have a rescue-worker reserved for execution under memory pressure. Else it is possible that the worker-pool deadlocks waiting for execution contexts to free up.

1.2.4 Application Programming Interface (API)

`alloc_workqueue()` allocates a wq. The original `create_*workqueue()` functions are deprecated and scheduled for removal. `alloc_workqueue()` takes three arguments - `@name`, `@flags` and `@max_active`. `@name` is the name of the wq and also used as the name of the rescuer thread if there is one.

A wq no longer manages execution resources but serves as a domain for forward progress guarantee, flush and work item attributes. `@flags` and `@max_active` control how work items are assigned execution resources, scheduled and executed.

`flags`

`WQ_UNBOUND`
Work items queued to an unbound wq are served by the special worker-pools which host workers which are not bound to any specific CPU. This makes the wq behave as a simple execution context provider without concurrency management. The unbound worker-pools try to start execution of work items as soon as possible. Unbound wq sacrifices locality but is useful for the following cases.

- Wide fluctuation in the concurrency level requirement is expected and using bound wq may end up creating large number of mostly unused workers across different CPUs as the issuer hops through different CPUs.
- Long running CPU intensive workloads which can be better managed by the system scheduler.

`WQ_FREEZABLE`
A freezable wq participates in the freeze phase of the system suspend operations. Work items on the wq are drained and no new work item starts execution until thawed.

`WQ_MEM_RECLAIM`
All wq which might be used in the memory reclaim paths **MUST** have this flag set. The wq is guaranteed to have at least one execution context regardless of memory pressure.

`WQ_HIGHPRI`
Work items of a highpri wq are queued to the highpri worker-pool of the target cpu. Highpri worker-pools are served by worker threads with elevated nice level.

Note that normal and highpri worker-pools don't interact with each other. Each maintains its separate pool of workers and implements concurrency management among its workers.

`WQ_CPU_INTENSIVE`
Work items of a CPU intensive wq do not contribute to the concurrency level. In other

words, runnable CPU intensive work items will not prevent other work items in the same worker-pool from starting execution. This is useful for bound work items which are expected to hog CPU cycles so that their execution is regulated by the system scheduler.

Although CPU intensive work items don't contribute to the concurrency level, start of their executions is still regulated by the concurrency management and runnable non-CPU-intensive work items can delay execution of CPU intensive work items.

This flag is meaningless for unbound wq.

max_active

`@max_active` determines the maximum number of execution contexts per CPU which can be assigned to the work items of a wq. For example, with `@max_active` of 16, at most 16 work items of the wq can be executing at the same time per CPU. This is always a per-CPU attribute, even for unbound workqueues.

The maximum limit for `@max_active` is 512 and the default value used when 0 is specified is 256. These values are chosen sufficiently high such that they are not the limiting factor while providing protection in runaway cases.

The number of active work items of a wq is usually regulated by the users of the wq, more specifically, by how many work items the users may queue at the same time. Unless there is a specific need for throttling the number of active work items, specifying '0' is recommended.

Some users depend on the strict execution ordering of ST wq. The combination of `@max_active` of 1 and `WQ_UNBOUND` used to achieve this behavior. Work items on such wq were always queued to the unbound worker-pools and only one work item could be active at any given time thus achieving the same ordering property as ST wq.

In the current implementation the above configuration only guarantees ST behavior within a given NUMA node. Instead `alloc_ordered_workqueue()` should be used to achieve system-wide ST behavior.

1.2.5 Example Execution Scenarios

The following example execution scenarios try to illustrate how cmwq behave under different configurations.

> Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU. w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms again before finishing. w1 and w2 burn CPU for 5ms then sleep for 10ms.

Ignoring all other tasks, works and processing overhead, and assuming simple FIFO scheduling, the following is one highly simplified version of possible sequences of events with the original wq.

```
TIME IN MSECS  EVENT
0              w0 starts and burns CPU
5              w0 sleeps
15             w0 wakes up and burns CPU
20             w0 finishes
20             w1 starts and burns CPU
25             w1 sleeps
```

```
35              w1 wakes up and finishes
35              w2 starts and burns CPU
40              w2 sleeps
50              w2 wakes up and finishes
```

And with cmwq with `@max_active` >= 3,

```
TIME IN MSECS  EVENT
0              w0 starts and burns CPU
5              w0 sleeps
5              w1 starts and burns CPU
10             w1 sleeps
10             w2 starts and burns CPU
15             w2 sleeps
15             w0 wakes up and burns CPU
20             w0 finishes
20             w1 wakes up and finishes
25             w2 wakes up and finishes
```

If `@max_active` == 2,

```
TIME IN MSECS  EVENT
0              w0 starts and burns CPU
5              w0 sleeps
5              w1 starts and burns CPU
10             w1 sleeps
15             w0 wakes up and burns CPU
20             w0 finishes
20             w1 wakes up and finishes
20             w2 starts and burns CPU
25             w2 sleeps
35             w2 wakes up and finishes
```

Now, let's assume w1 and w2 are queued to a different wq q1 which has `WQ_CPU_INTENSIVE` set,

```
TIME IN MSECS  EVENT
0              w0 starts and burns CPU
5              w0 sleeps
5              w1 and w2 start and burn CPU
10             w1 sleeps
15             w2 sleeps
15             w0 wakes up and burns CPU
20             w0 finishes
20             w1 wakes up and finishes
25             w2 wakes up and finishes
```

1.2.6 Guidelines

- Do not forget to use `WQ_MEM_RECLAIM` if a wq may process work items which are used during memory reclaim. Each wq with `WQ_MEM_RECLAIM` set has an execution context reserved for it. If there is dependency among multiple work items used during memory reclaim, they should be queued to separate wq each with `WQ_MEM_RECLAIM`.
- Unless strict ordering is required, there is no need to use ST wq.
- Unless there is a specific need, using 0 for @max_active is recommended. In most use cases, concurrency level usually stays well under the default limit.
- A wq serves as a domain for forward progress guarantee (`WQ_MEM_RECLAIM`, flush and work item attributes. Work items which are not involved in memory reclaim and don't need to be flushed as a part of a group of work items, and don't require any special attribute, can use one of the system wq. There is no difference in execution characteristics between using a dedicated wq and a system wq.
- Unless work items are expected to consume a huge amount of CPU cycles, using a bound wq is usually beneficial due to the increased level of locality in wq operations and work item execution.

1.2.7 Affinity Scopes

An unbound workqueue groups CPUs according to its affinity scope to improve cache locality. For example, if a workqueue is using the default affinity scope of "cache", it will group CPUs according to last level cache boundaries. A work item queued on the workqueue will be assigned to a worker on one of the CPUs which share the last level cache with the issuing CPU. Once started, the worker may or may not be allowed to move outside the scope depending on the `affinity_strict` setting of the scope.

Workqueue currently supports the following affinity scopes.

default
: Use the scope in module parameter `workqueue.default_affinity_scope` which is always set to one of the scopes below.

cpu
: CPUs are not grouped. A work item issued on one CPU is processed by a worker on the same CPU. This makes unbound workqueues behave as per-cpu workqueues without concurrency management.

smt
: CPUs are grouped according to SMT boundaries. This usually means that the logical threads of each physical CPU core are grouped together.

cache
: CPUs are grouped according to cache boundaries. Which specific cache boundary is used is determined by the arch code. L3 is used in a lot of cases. This is the default affinity scope.

numa
: CPUs are grouped according to NUMA bounaries.

system
All CPUs are put in the same group. Workqueue makes no effort to process a work item on a CPU close to the issuing CPU.

The default affinity scope can be changed with the module parameter `workqueue.default_affinity_scope` and a specific workqueue's affinity scope can be changed using `apply_workqueue_attrs()`.

If `WQ_SYSFS` is set, the workqueue will have the following affinity scope related interface files under its `/sys/devices/virtual/workqueue/WQ_NAME/` directory.

affinity_scope
Read to see the current affinity scope. Write to change.

When default is the current scope, reading this file will also show the current effective scope in parentheses, for example, `default (cache)`.

affinity_strict
0 by default indicating that affinity scopes are not strict. When a work item starts execution, workqueue makes a best-effort attempt to ensure that the worker is inside its affinity scope, which is called repatriation. Once started, the scheduler is free to move the worker anywhere in the system as it sees fit. This enables benefiting from scope locality while still being able to utilize other CPUs if necessary and available.

If set to 1, all workers of the scope are guaranteed always to be in the scope. This may be useful when crossing affinity scopes has other implications, for example, in terms of power consumption or workload isolation. Strict NUMA scope can also be used to match the workqueue behavior of older kernels.

1.2.8 Affinity Scopes and Performance

It'd be ideal if an unbound workqueue's behavior is optimal for vast majority of use cases without further tuning. Unfortunately, in the current kernel, there exists a pronounced trade-off between locality and utilization necessitating explicit configurations when workqueues are heavily used.

Higher locality leads to higher efficiency where more work is performed for the same number of consumed CPU cycles. However, higher locality may also cause lower overall system utilization if the work items are not spread enough across the affinity scopes by the issuers. The following performance testing with dm-crypt clearly illustrates this trade-off.

The tests are run on a CPU with 12-cores/24-threads split across four L3 caches (AMD Ryzen 9 3900x). CPU clock boost is turned off for consistency. `/dev/dm-0` is a dm-crypt device created on NVME SSD (Samsung 990 PRO) and opened with `cryptsetup` with default settings.

Scenario 1: Enough issuers and work spread across the machine

The command used:

```
$ fio --filename=/dev/dm-0 --direct=1 --rw=randrw --bs=32k --ioengine=libaio \
  --iodepth=64 --runtime=60 --numjobs=24 --time_based --group_reporting \
  --name=iops-test-job --verify=sha512
```

There are 24 issuers, each issuing 64 IOs concurrently. `--verify=sha512` makes `fio` generate and read back the content each time which makes execution locality matter between the issuer and `kcryptd`. The followings are the read bandwidths and CPU utilizations depending on different affinity scope settings on `kcryptd` measured over five runs. Bandwidths are in MiBps, and CPU util in percents.

Affinity	Bandwidth (MiBps)	CPU util (%)
system	1159.40 ±1.34	99.31 ±0.02
cache	1166.40 ±0.89	99.34 ±0.01
cache (strict)	1166.00 ±0.71	99.35 ±0.01

With enough issuers spread across the system, there is no downside to "cache", strict or otherwise. All three configurations saturate the whole machine but the cache-affine ones outperform by 0.6% thanks to improved locality.

Scenario 2: Fewer issuers, enough work for saturation

The command used:

```
$ fio --filename=/dev/dm-0 --direct=1 --rw=randrw --bs=32k \
  --ioengine=libaio --iodepth=64 --runtime=60 --numjobs=8 \
  --time_based --group_reporting --name=iops-test-job --verify=sha512
```

The only difference from the previous scenario is `--numjobs=8`. There are a third of the issuers but is still enough total work to saturate the system.

Affinity	Bandwidth (MiBps)	CPU util (%)
system	1155.40 ±0.89	97.41 ±0.05
cache	1154.40 ±1.14	96.15 ±0.09
cache (strict)	1112.00 ±4.64	93.26 ±0.35

This is more than enough work to saturate the system. Both "system" and "cache" are nearly saturating the machine but not fully. "cache" is using less CPU but the better efficiency puts it at the same bandwidth as "system".

Eight issuers moving around over four L3 cache scope still allow "cache (strict)" to mostly saturate the machine but the loss of work conservation is now starting to hurt with 3.7% bandwidth loss.

Scenario 3: Even fewer issuers, not enough work to saturate

The command used:

```
$ fio --filename=/dev/dm-0 --direct=1 --rw=randrw --bs=32k \
  --ioengine=libaio --iodepth=64 --runtime=60 --numjobs=4 \
  --time_based --group_reporting --name=iops-test-job --verify=sha512
```

Again, the only difference is `--numjobs=4`. With the number of issuers reduced to four, there now isn't enough work to saturate the whole system and the bandwidth becomes dependent on completion latencies.

Affinity	Bandwidth (MiBps)	CPU util (%)
system	993.60 ±1.82	75.49 ±0.06
cache	973.40 ±1.52	74.90 ±0.07
cache (strict)	828.20 ±4.49	66.84 ±0.29

Now, the tradeoff between locality and utilization is clearer. "cache" shows 2% bandwidth loss compared to "system" and "cache (struct)" whopping 20%.

Conclusion and Recommendations

In the above experiments, the efficiency advantage of the "cache" affinity scope over "system" is, while consistent and noticeable, small. However, the impact is dependent on the distances between the scopes and may be more pronounced in processors with more complex topologies.

While the loss of work-conservation in certain scenarios hurts, it is a lot better than "cache (strict)" and maximizing workqueue utilization is unlikely to be the common case anyway. As such, "cache" is the default affinity scope for unbound pools.

- As there is no one option which is great for most cases, workqueue usages that may consume a significant amount of CPU are recommended to configure the workqueues using `apply_workqueue_attrs()` and/or enable `WQ_SYSFS`.
- An unbound workqueue with strict "cpu" affinity scope behaves the same as `WQ_CPU_INTENSIVE` per-cpu workqueue. There is no real advanage to the latter and an unbound workqueue provides a lot more flexibility.
- Affinity scopes are introduced in Linux v6.5. To emulate the previous behavior, use strict "numa" affinity scope.
- The loss of work-conservation in non-strict affinity scopes is likely originating from the scheduler. There is no theoretical reason why the kernel wouldn't be able to do the right thing and maintain work-conservation in most cases. As such, it is possible that future scheduler improvements may make most of these tunables unnecessary.

1.2.9 Examining Configuration

Use tools/workqueue/wq_dump.py to examine unbound CPU affinity configuration, worker pools and how workqueues map to the pools:

```
$ tools/workqueue/wq_dump.py
Affinity Scopes
===============
wq_unbound_cpumask=0000000f

CPU
  nr_pods  4
  pod_cpus [0]=00000001 [1]=00000002 [2]=00000004 [3]=00000008
  pod_node [0]=0 [1]=0 [2]=1 [3]=1
  cpu_pod  [0]=0 [1]=1 [2]=2 [3]=3

SMT
  nr_pods  4
  pod_cpus [0]=00000001 [1]=00000002 [2]=00000004 [3]=00000008
  pod_node [0]=0 [1]=0 [2]=1 [3]=1
  cpu_pod  [0]=0 [1]=1 [2]=2 [3]=3

CACHE (default)
  nr_pods  2
  pod_cpus [0]=00000003 [1]=0000000c
  pod_node [0]=0 [1]=1
  cpu_pod  [0]=0 [1]=0 [2]=1 [3]=1

NUMA
  nr_pods  2
  pod_cpus [0]=00000003 [1]=0000000c
  pod_node [0]=0 [1]=1
  cpu_pod  [0]=0 [1]=0 [2]=1 [3]=1

SYSTEM
  nr_pods  1
  pod_cpus [0]=0000000f
  pod_node [0]=-1
  cpu_pod  [0]=0 [1]=0 [2]=0 [3]=0

Worker Pools
============
pool[00] ref= 1 nice=  0 idle/workers=  4/  4 cpu=  0
pool[01] ref= 1 nice=-20 idle/workers=  2/  2 cpu=  0
pool[02] ref= 1 nice=  0 idle/workers=  4/  4 cpu=  1
pool[03] ref= 1 nice=-20 idle/workers=  2/  2 cpu=  1
pool[04] ref= 1 nice=  0 idle/workers=  4/  4 cpu=  2
pool[05] ref= 1 nice=-20 idle/workers=  2/  2 cpu=  2
pool[06] ref= 1 nice=  0 idle/workers=  3/  3 cpu=  3
pool[07] ref= 1 nice=-20 idle/workers=  2/  2 cpu=  3
pool[08] ref=42 nice=  0 idle/workers=  6/  6 cpus=0000000f
```

```
pool[09] ref=28 nice=  0 idle/workers=  3/  3 cpus=00000003
pool[10] ref=28 nice=  0 idle/workers= 17/ 17 cpus=0000000c
pool[11] ref= 1 nice=-20 idle/workers=  1/  1 cpus=0000000f
pool[12] ref= 2 nice=-20 idle/workers=  1/  1 cpus=00000003
pool[13] ref= 2 nice=-20 idle/workers=  1/  1 cpus=0000000c

Workqueue CPU -> pool
=====================
[    workqueue \ CPU              0  1  2  3 dfl]
events                   percpu   0  2  4  6
events_highpri           percpu   1  3  5  7
events_long              percpu   0  2  4  6
events_unbound           unbound  9  9 10 10  8
events_freezable         percpu   0  2  4  6
events_power_efficient   percpu   0  2  4  6
events_freezable_power_  percpu   0  2  4  6
rcu_gp                   percpu   0  2  4  6
rcu_par_gp               percpu   0  2  4  6
slub_flushwq             percpu   0  2  4  6
netns                    ordered  8  8  8  8  8
...
```

See the command's help message for more info.

1.2.10 Monitoring

Use tools/workqueue/wq_monitor.py to monitor workqueue operations:

```
$ tools/workqueue/wq_monitor.py events
                            total  infl  CPUtime  CPUhog CMW/RPR  mayday␣
↪rescued
events                      18545     0      6.1       0       5       -      ␣
↪-
events_highpri                  8     0      0.0       0       0       -      ␣
↪-
events_long                     3     0      0.0       0       0       -      ␣
↪-
events_unbound              38306     0      0.1       -       7       -      ␣
↪-
events_freezable                0     0      0.0       0       0       -      ␣
↪-
events_power_efficient      29598     0      0.2       0       0       -      ␣
↪-
events_freezable_power_        10     0      0.0       0       0       -      ␣
↪-
sock_diag_events                0     0      0.0       0       0       -      ␣
↪-

                            total  infl  CPUtime  CPUhog CMW/RPR  mayday␣
↪rescued
```

```
events                    18548     0       6.1       0       5       -      ␣
↪-
events_highpri                8     0       0.0       0       0       -      ␣
↪-
events_long                   3     0       0.0       0       0       -      ␣
↪-
events_unbound            38322     0       0.1       -       7       -      ␣
↪-
events_freezable              0     0       0.0       0       0       -      ␣
↪-
events_power_efficient    29603     0       0.2       0       0       -      ␣
↪-
events_freezable_power_      10     0       0.0       0       0       -      ␣
↪-
sock_diag_events              0     0       0.0       0       0       -      ␣
↪-

...
```

See the command's help message for more info.

1.2.11 Debugging

Because the work functions are executed by generic worker threads there are a few tricks needed to shed some light on misbehaving workqueue users.

Worker threads show up in the process list as:

```
root      5671  0.0  0.0      0     0 ?        S    12:07   0:00 [kworker/0:1]
root      5672  0.0  0.0      0     0 ?        S    12:07   0:00 [kworker/1:2]
root      5673  0.0  0.0      0     0 ?        S    12:12   0:00 [kworker/0:0]
root      5674  0.0  0.0      0     0 ?        S    12:13   0:00 [kworker/1:0]
```

If kworkers are going crazy (using too much cpu), there are two types of possible problems:

1. Something being scheduled in rapid succession
2. A single work item that consumes lots of cpu cycles

The first one can be tracked using tracing:

```
$ echo workqueue:workqueue_queue_work > /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace_pipe > out.txt
(wait a few secs)
^C
```

If something is busy looping on work queueing, it would be dominating the output and the offender can be determined with the work item function.

For the second type of problems it should be possible to just check the stack trace of the offending worker thread.

```
$ cat /proc/THE_OFFENDING_KWORKER/stack
```

The work item's function should be trivially visible in the stack trace.

1.2.12 Non-reentrance Conditions

Workqueue guarantees that a work item cannot be re-entrant if the following conditions hold after a work item gets queued:

1. The work function hasn't been changed.
2. No one queues the work item to another workqueue.
3. The work item hasn't been reinitiated.

In other words, if the above conditions hold, the work item is guaranteed to be executed by at most one worker system-wide at any given time.

Note that requeuing the work item (to the same queue) in the self function doesn't break these conditions, so it's safe to do. Otherwise, caution is required when breaking the conditions inside a work function.

1.2.13 Kernel Inline Documentations Reference

struct **workqueue_attrs**

A struct for workqueue attributes.

Definition:

```
struct workqueue_attrs {
    int nice;
    cpumask_var_t cpumask;
    cpumask_var_t __pod_cpumask;
    bool affn_strict;
    enum wq_affn_scope affn_scope;
    bool ordered;
};
```

Members

nice
nice level

cpumask
allowed CPUs

Work items in this workqueue are affine to these CPUs and not allowed to execute on other CPUs. A pool serving a workqueue must have the same **cpumask**.

__pod_cpumask
internal attribute used to create per-pod pools

Internal use only.

Per-pod unbound worker pools are used to improve locality. Always a subset of ->cpumask. A workqueue can be associated with multiple worker pools with disjoint **__pod_cpumask**'s. Whether the enforcement of a pool's **__pod_cpumask** is strict depends on **affn_strict**.

affn_strict
affinity scope is strict

If clear, workqueue will make a best-effort attempt at starting the worker inside **__pod_cpumask** but the scheduler is free to migrate it outside.

If set, workers are only allowed to run inside **__pod_cpumask**.

affn_scope
unbound CPU affinity scope

CPU pods are used to improve execution locality of unbound work items. There are multiple pod types, one for each wq_affn_scope, and every CPU in the system belongs to one pod in every pod type. CPUs that belong to the same pod share the worker pool. For example, selecting `WQ_AFFN_NUMA` makes the workqueue use a separate worker pool for each NUMA node.

ordered
work items must be executed one by one in queueing order

Description

This can be used to change attributes of an unbound workqueue.

work_pending

`work_pending (work)`

Find out whether a work item is currently pending

Parameters

work
The work item in question

delayed_work_pending

`delayed_work_pending (w)`

Find out whether a delayable work item is currently pending

Parameters

w
The work item in question

struct workqueue_struct ***alloc_workqueue**(const char *fmt, unsigned int flags, int max_active, ...)

allocate a workqueue

Parameters

const char *fmt
printf format for the name of the workqueue

unsigned int flags
WQ_* flags

int max_active
max in-flight work items per CPU, 0 for default remaining args: args for **fmt**

...
variable arguments

Description

Allocate a workqueue with the specified parameters. For detailed information on WQ_* flags, please refer to *Workqueue*.

Return

Pointer to the allocated workqueue on success, `NULL` on failure.

alloc_ordered_workqueue

`alloc_ordered_workqueue (fmt, flags, args...)`

allocate an ordered workqueue

Parameters

fmt
printf format for the name of the workqueue

flags
WQ_* flags (only WQ_FREEZABLE and WQ_MEM_RECLAIM are meaningful)

args...
args for **fmt**

Description

Allocate an ordered workqueue. An ordered workqueue executes at most one work item at any given time in the queued order. They are implemented as unbound workqueues with **max_active** of one.

Return

Pointer to the allocated workqueue on success, `NULL` on failure.

bool **queue_work**(struct workqueue_struct *wq, struct work_struct *work)

queue work on a workqueue

Parameters

struct workqueue_struct *wq
workqueue to use

struct work_struct *work
work to queue

Description

Returns `false` if **work** was already on a queue, `true` otherwise.

We queue the work to the CPU on which it was submitted, but if the CPU dies it can be processed by another CPU.

Memory-ordering properties: If it returns `true`, guarantees that all stores preceding the call to *queue_work()* in the program order will be visible from the CPU which will execute **work** by the time such work executes, e.g.,

{ x is initially 0 }

CPU0 CPU1

WRITE_ONCE(x, 1); [**work** is being executed] r0 = queue_work(wq, work); r1 = READ_ONCE(x);

Forbids: r0 == true && r1 == 0

bool **queue_delayed_work**(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

queue work on a workqueue after delay

Parameters

struct workqueue_struct *wq
workqueue to use

struct delayed_work *dwork
delayable work to queue

unsigned long delay
number of jiffies to wait before queueing

Description

Equivalent to *queue_delayed_work_on()* but tries to use the local CPU.

bool **mod_delayed_work**(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

modify delay of or queue a delayed work

Parameters

struct workqueue_struct *wq
workqueue to use

struct delayed_work *dwork
work to queue

unsigned long delay
number of jiffies to wait before queueing

Description

mod_delayed_work_on() on local CPU.

bool **schedule_work_on**(int cpu, struct work_struct *work)

put work task on a specific cpu

Parameters

int cpu
cpu to put the work task on

struct work_struct *work
job to be done

Description

This puts a job on a specific cpu

bool **schedule_work**(struct work_struct *work)
    put work task in global workqueue

Parameters

struct work_struct *work
    job to be done

Description

Returns `false` if **work** was already on the kernel-global workqueue and `true` otherwise.

This puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

Shares the same memory-ordering properties of *queue_work()*, cf. the DocBook header of *queue_work()*.

bool **schedule_delayed_work_on**(int cpu, struct delayed_work *dwork, unsigned long delay)
    queue work in global workqueue on CPU after delay

Parameters

int cpu
    cpu to use

struct delayed_work *dwork
    job to be done

unsigned long delay
    number of jiffies to wait

Description

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

bool **schedule_delayed_work**(struct delayed_work *dwork, unsigned long delay)
    put work task in global workqueue after delay

Parameters

struct delayed_work *dwork
    job to be done

unsigned long delay
    number of jiffies to wait or 0 for immediate execution

Description

After waiting for a given time this puts a job in the kernel-global workqueue.

for_each_pool

`for_each_pool (pool, pi)`

    iterate through all worker_pools in the system

Parameters

pool
    iteration cursor

pi
integer used for iteration

Description

This must be called either with wq_pool_mutex held or RCU read locked. If the pool needs to be used beyond the locking in effect, the caller is responsible for guaranteeing that the pool stays online.

The if/else clause exists only for the lockdep assertion and can be ignored.

for_each_pool_worker

`for_each_pool_worker (worker, pool)`

iterate through all workers of a worker_pool

Parameters

worker
iteration cursor

pool
worker_pool to iterate workers of

Description

This must be called with wq_pool_attach_mutex.

The if/else clause exists only for the lockdep assertion and can be ignored.

for_each_pwq

`for_each_pwq (pwq, wq)`

iterate through all pool_workqueues of the specified workqueue

Parameters

pwq
iteration cursor

wq
the target workqueue

Description

This must be called either with wq->mutex held or RCU read locked. If the pwq needs to be used beyond the locking in effect, the caller is responsible for guaranteeing that the pwq stays online.

The if/else clause exists only for the lockdep assertion and can be ignored.

int **worker_pool_assign_id**(struct worker_pool *pool)

allocate ID and assign it to **pool**

Parameters

struct worker_pool *pool
the pool pointer of interest

Description

Returns 0 if ID in [0, WORK_OFFQ_POOL_NONE) is allocated and assigned successfully, -errno on failure.

struct worker_pool ***get_work_pool**(struct work_struct *work)
return the worker_pool a given work was associated with

Parameters

struct work_struct *work
the work item of interest

Description

Pools are created and destroyed under wq_pool_mutex, and allows read access under RCU read lock. As such, this function should be called under wq_pool_mutex or inside of a rcu_read_lock() region.

All fields of the returned pool are accessible as long as the above mentioned locking is in effect. If the returned pool needs to be used beyond the critical section, the caller is responsible for ensuring the returned pool is and stays online.

Return

The worker_pool **work** was last associated with. `NULL` if none.

int **get_work_pool_id**(struct work_struct *work)
return the worker pool ID a given work is associated with

Parameters

struct work_struct *work
the work item of interest

Return

The worker_pool ID **work** was last associated with. `WORK_OFFQ_POOL_NONE` if none.

void **worker_set_flags**(struct *worker* *worker, unsigned int flags)
set worker flags and adjust nr_running accordingly

Parameters

struct worker *worker
self

unsigned int flags
flags to set

Description

Set **flags** in **worker->flags** and adjust nr_running accordingly.

void **worker_clr_flags**(struct *worker* *worker, unsigned int flags)
clear worker flags and adjust nr_running accordingly

Parameters

struct worker *worker
self

unsigned int flags
flags to clear

Description

Clear **flags** in **worker->flags** and adjust nr_running accordingly.

void **worker_enter_idle**(struct *worker* *worker)
enter idle state

Parameters

struct worker *worker
worker which is entering idle state

Description

worker is entering idle state. Update stats and idle timer if necessary.

LOCKING: raw_spin_lock_irq(pool->lock).

void **worker_leave_idle**(struct *worker* *worker)
leave idle state

Parameters

struct worker *worker
worker which is leaving idle state

Description

worker is leaving idle state. Update stats.

LOCKING: raw_spin_lock_irq(pool->lock).

struct worker ***find_worker_executing_work**(struct worker_pool *pool, struct work_struct *work)
find worker which is executing a work

Parameters

struct worker_pool *pool
pool of interest

struct work_struct *work
work to find worker for

Description

Find a worker which is executing **work** on **pool** by searching **pool->busy_hash** which is keyed by the address of **work**. For a worker to match, its current execution should match the address of **work** and its work function. This is to avoid unwanted dependency between unrelated work executions through a work item being recycled while still being executed.

This is a bit tricky. A work item may be freed once its execution starts and nothing prevents the freed area from being recycled for another work item. If the same work item address ends up being reused before the original execution finishes, workqueue will identify the recycled work item as currently executing and make it wait until the current execution finishes, introducing an unwanted dependency.

This function checks the work item address and work function to avoid false positives. Note that this isn't complete as one may construct a work function which can introduce dependency onto itself through a recycled work item. Well, if somebody wants to shoot oneself in the foot that badly, there's only so much we can do, and if such deadlock actually occurs, it should be easy to locate the culprit work function.

Context

raw_spin_lock_irq(pool->lock).

Return

Pointer to worker which is executing **work** if found, `NULL` otherwise.

void **move_linked_works**(struct work_struct *work, struct list_head *head, struct work_struct **nextp)

move linked works to a list

Parameters

struct work_struct *work
start of series of works to be scheduled

struct list_head *head
target list to append **work** to

struct work_struct **nextp
out parameter for nested worklist walking

Description

Schedule linked works starting from **work** to **head**. Work series to be scheduled starts at **work** and includes any consecutive work with WORK_STRUCT_LINKED set in its predecessor. See *assign_work()* for details on **nextp**.

Context

raw_spin_lock_irq(pool->lock).

bool **assign_work**(struct work_struct *work, struct *worker* *worker, struct work_struct **nextp)

assign a work item and its linked work items to a worker

Parameters

struct work_struct *work
work to assign

struct worker *worker
worker to assign to

struct work_struct **nextp
out parameter for nested worklist walking

Description

Assign **work** and its linked work items to **worker**. If **work** is already being executed by another worker in the same pool, it'll be punted there.

If **nextp** is not NULL, it's updated to point to the next work of the last scheduled work. This allows *assign_work()* to be nested inside list_for_each_entry_safe().

Returns `true` if **work** was successfully assigned to **worker**. `false` if **work** was punted to another worker already executing it.

bool **kick_pool**(struct worker_pool *pool)
wake up an idle worker if necessary

Parameters

struct worker_pool *pool
pool to kick

Description

pool may have pending work items. Wake up worker if necessary. Returns whether a worker was woken up.

void **wq_worker_running**(struct task_struct *task)
a worker is running again

Parameters

struct task_struct *task
task waking up

Description

This function is called when a worker returns from schedule()

void **wq_worker_sleeping**(struct task_struct *task)
a worker is going to sleep

Parameters

struct task_struct *task
task going to sleep

Description

This function is called from schedule() when a busy worker is going to sleep.

void **wq_worker_tick**(struct task_struct *task)
a scheduler tick occurred while a kworker is running

Parameters

struct task_struct *task
task currently running

Description

Called from scheduler_tick(). We're in the IRQ context and the current worker's fields which follow the 'K' locking rule can be accessed safely.

work_func_t **wq_worker_last_func**(struct task_struct *task)
retrieve worker's last work function

Parameters

struct task_struct *task
Task to retrieve last work function of.

Description

Determine the last function a worker executed. This is called from the scheduler to get a worker's last known identity.

This function is called during schedule() when a kworker is going to sleep. It's used by psi to identify aggregation workers during dequeuing, to allow periodic aggregation to shut-off when that worker is the last task in the system or cgroup to go to sleep.

As this function doesn't involve any workqueue-related locking, it only returns stable values when called from inside the scheduler's queuing and dequeuing paths, when **task**, which must be a kworker, is guaranteed to not be processing any works.

Context

raw_spin_lock_irq(rq->lock)

Return

The last work function `current` executed as a worker, NULL if it hasn't executed any work yet.

void **get_pwq**(struct pool_workqueue *pwq)

get an extra reference on the specified pool_workqueue

Parameters

struct pool_workqueue *pwq
pool_workqueue to get

Description

Obtain an extra reference on **pwq**. The caller should guarantee that **pwq** has positive refcnt and be holding the matching pool->lock.

void **put_pwq**(struct pool_workqueue *pwq)

put a pool_workqueue reference

Parameters

struct pool_workqueue *pwq
pool_workqueue to put

Description

Drop a reference of **pwq**. If its refcnt reaches zero, schedule its destruction. The caller should be holding the matching pool->lock.

void **put_pwq_unlocked**(struct pool_workqueue *pwq)

put_pwq() with surrounding pool lock/unlock

Parameters

struct pool_workqueue *pwq
pool_workqueue to put (can be `NULL`)

Description

put_pwq() with locking. This function also allows `NULL` **pwq**.

void **pwq_dec_nr_in_flight**(struct pool_workqueue *pwq, unsigned long work_data)

decrement pwq's nr_in_flight

Parameters

struct pool_workqueue *pwq
pwq of interest

unsigned long work_data
work_data of work which left the queue

Description

A work either has completed or is removed from pending queue, decrement nr_in_flight of its pwq and handle workqueue flushing.

Context

raw_spin_lock_irq(pool->lock).

int **try_to_grab_pending**(struct work_struct *work, bool is_dwork, unsigned long *flags)
steal work item from worklist and disable irq

Parameters

struct work_struct *work
work item to steal

bool is_dwork
work is a delayed_work

unsigned long *flags
place to store irq state

Description

Try to grab PENDING bit of **work**. This function can handle **work** in any stable state - idle, on timer or on worklist.

On successful return, >= 0, irq is disabled and the caller is responsible for releasing it using local_irq_restore(***flags**).

This function is safe to call from any context including IRQ handler.

Return

1	if **work** was pending and we successfully stole PENDING
0	if **work** was idle and we claimed PENDING
-EAGAIN	if PENDING couldn't be grabbed at the moment, safe to busy-retry
-ENOENT	if someone else is canceling **work**, this state may persist for arbitrarily long

Note

On >= 0 return, the caller owns **work**'s PENDING bit. To avoid getting interrupted while holding PENDING and **work** off queue, irq must be disabled on entry. This, combined with delayed_work->timer being irqsafe, ensures that we return -EAGAIN for finite short period of time.

void **insert_work**(struct pool_workqueue *pwq, struct work_struct *work, struct list_head *head, unsigned int extra_flags)
insert a work into a pool

Parameters

struct pool_workqueue *pwq
pwq **work** belongs to

struct work_struct *work
work to insert

struct list_head *head
insertion point

unsigned int extra_flags
extra WORK_STRUCT_* flags to set

Description

Insert **work** which belongs to **pwq** after **head**. **extra_flags** is or'd to work_struct flags.

Context

raw_spin_lock_irq(pool->lock).

bool **queue_work_on**(int cpu, struct workqueue_struct *wq, struct work_struct *work)
queue work on specific cpu

Parameters

int cpu
CPU number to execute work on

struct workqueue_struct *wq
workqueue to use

struct work_struct *work
work to queue

Description

We queue the work to a specific CPU, the caller must ensure it can't go away. Callers that fail to ensure that the specified CPU cannot go away will execute on a randomly chosen CPU. But note well that callers specifying a CPU that never has been online will get a splat.

Return

`false` if **work** was already on a queue, `true` otherwise.

int **select_numa_node_cpu**(int node)
Select a CPU based on NUMA node

Parameters

int node
NUMA node ID that we want to select a CPU from

Description

This function will attempt to find a "random" cpu available on a given node. If there are no CPUs available on the given node it will return WORK_CPU_UNBOUND indicating that we should just schedule to any available CPU if we need to schedule this work.

bool **queue_work_node**(int node, struct workqueue_struct *wq, struct work_struct *work)
queue work on a "random" cpu for a given NUMA node

Parameters

int node
NUMA node that we are targeting the work for

struct workqueue_struct *wq
workqueue to use

struct work_struct *work
work to queue

Description

We queue the work to a "random" CPU within a given NUMA node. The basic idea here is to provide a way to somehow associate work with a given NUMA node.

This function will only make a best effort attempt at getting this onto the right NUMA node. If no node is requested or the requested node is offline then we just fall back to standard queue_work behavior.

Currently the "random" CPU ends up being the first available CPU in the intersection of cpu_online_mask and the cpumask of the node, unless we are running on the node. In that case we just use the current CPU.

Return

`false` if **work** was already on a queue, `true` otherwise.

bool **queue_delayed_work_on**(int cpu, struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

queue work on specific CPU after delay

Parameters

int cpu
CPU number to execute work on

struct workqueue_struct *wq
workqueue to use

struct delayed_work *dwork
work to queue

unsigned long delay
number of jiffies to wait before queueing

Return

`false` if **work** was already on a queue, `true` otherwise. If **delay** is zero and **dwork** is idle, it will be scheduled for immediate execution.

bool **mod_delayed_work_on**(int cpu, struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

modify delay of or queue a delayed work on specific CPU

Parameters

int cpu
CPU number to execute work on

struct workqueue_struct *wq
workqueue to use

struct delayed_work *dwork
work to queue

unsigned long delay
number of jiffies to wait before queueing

Description

If **dwork** is idle, equivalent to *queue_delayed_work_on()*; otherwise, modify **dwork**'s timer so that it expires after **delay**. If **delay** is zero, **work** is guaranteed to be scheduled immediately regardless of its current state.

This function is safe to call from any context including IRQ handler. See *try_to_grab_pending()* for details.

Return

`false` if **dwork** was idle and queued, `true` if **dwork** was pending and its timer was modified.

bool **queue_rcu_work**(struct workqueue_struct *wq, struct rcu_work *rwork)
queue work after a RCU grace period

Parameters

struct workqueue_struct *wq
workqueue to use

struct rcu_work *rwork
work to queue

Return

`false` if **rwork** was already pending, `true` otherwise. Note that a full RCU grace period is guaranteed only after a `true` return. While **rwork** is guaranteed to be executed after a `false` return, the execution may happen before a full RCU grace period has passed.

void **worker_attach_to_pool**(struct *worker* *worker, struct worker_pool *pool)
attach a worker to a pool

Parameters

struct worker *worker
worker to be attached

struct worker_pool *pool
the target pool

Description

Attach **worker** to **pool**. Once attached, the `WORKER_UNBOUND` flag and cpu-binding of **worker** are kept coordinated with the pool across cpu-[un]hotplugs.

void **worker_detach_from_pool**(struct *worker* *worker)
detach a worker from its pool

Parameters

struct worker *worker
worker which is attached to its pool

Description

Undo the attaching which had been done in *worker_attach_to_pool()*. The caller worker shouldn't access to the pool after detached except it has other reference to the pool.

struct worker ***create_worker**(struct worker_pool *pool)

create a new workqueue worker

Parameters

struct worker_pool *pool
pool the new worker will belong to

Description

Create and start a new worker which is attached to **pool**.

Context

Might sleep. Does GFP_KERNEL allocations.

Return

Pointer to the newly created worker.

void **set_worker_dying**(struct *worker* *worker, struct list_head *list)

Tag a worker for destruction

Parameters

struct worker *worker
worker to be destroyed

struct list_head *list
transfer worker away from its pool->idle_list and into list

Description

Tag **worker** for destruction and adjust **pool** stats accordingly. The worker should be idle.

Context

raw_spin_lock_irq(pool->lock).

void **idle_worker_timeout**(struct timer_list *t)

check if some idle workers can now be deleted.

Parameters

struct timer_list *t
The pool's idle_timer that just expired

Description

The timer is armed in *worker_enter_idle()*. Note that it isn't disarmed in *worker_leave_idle()*, as a worker flicking between idle and active while its pool is at the too_many_workers() tipping point would cause too much timer housekeeping overhead. Since IDLE_WORKER_TIMEOUT is long enough, we just let it expire and re-evaluate things from there.

void **idle_cull_fn**(struct work_struct *work)

cull workers that have been idle for too long.

Parameters

struct work_struct *work
the pool's work for handling these idle workers

Description

This goes through a pool's idle workers and gets rid of those that have been idle for at least IDLE_WORKER_TIMEOUT seconds.

We don't want to disturb isolated CPUs because of a pcpu kworker being culled, so this also resets worker affinity. This requires a sleepable context, hence the split between timer callback and work item.

void **maybe_create_worker**(struct worker_pool *pool)
create a new worker if necessary

Parameters

struct worker_pool *pool
pool to create a new worker for

Description

Create a new worker for **pool** if necessary. **pool** is guaranteed to have at least one idle worker on return from this function. If creating a new worker takes longer than MAYDAY_INTERVAL, mayday is sent to all rescuers with works scheduled on **pool** to resolve possible allocation deadlock.

On return, need_to_create_worker() is guaranteed to be `false` and may_start_working() `true`.

LOCKING: raw_spin_lock_irq(pool->lock) which may be released and regrabbed multiple times. Does GFP_KERNEL allocations. Called only from manager.

bool **manage_workers**(struct *worker* *worker)
manage worker pool

Parameters

struct worker *worker
self

Description

Assume the manager role and manage the worker pool **worker** belongs to. At any given time, there can be only zero or one manager per pool. The exclusion is handled automatically by this function.

The caller can safely start processing works on false return. On true return, it's guaranteed that need_to_create_worker() is false and may_start_working() is true.

Context

raw_spin_lock_irq(pool->lock) which may be released and regrabbed multiple times. Does GFP_KERNEL allocations.

Return

`false` if the pool doesn't need management and the caller can safely start processing works, `true` if management function was performed and the conditions that the caller verified before calling the function may no longer be true.

void **process_one_work**(struct *worker* *worker, struct work_struct *work)
process single work

Parameters

struct worker *worker
self

struct work_struct *work
work to process

Description

Process **work**. This function contains all the logics necessary to process a single work including synchronization against and interaction with other workers on the same cpu, queueing and flushing. As long as context requirement is met, any worker can call this function to process a work.

Context

raw_spin_lock_irq(pool->lock) which is released and regrabbed.

void **process_scheduled_works**(struct *worker* *worker)
process scheduled works

Parameters

struct worker *worker
self

Description

Process all scheduled works. Please note that the scheduled list may change while processing a work, so this function repeatedly fetches a work from the top and executes it.

Context

raw_spin_lock_irq(pool->lock) which may be released and regrabbed multiple times.

int **worker_thread**(void *__worker)
the worker thread function

Parameters

void *__worker
self

Description

The worker thread function. All workers belong to a worker_pool - either a per-cpu one or dynamic unbound one. These workers process all work items regardless of their specific target workqueue. The only exception is work items which belong to workqueues with a rescuer which will be explained in *rescuer_thread()*.

Return

0

int **rescuer_thread**(void *__rescuer)
the rescuer thread function

Parameters

void *__rescuer
self

Description

Workqueue rescuer thread function. There's one rescuer for each workqueue which has WQ_MEM_RECLAIM set.

Regular work processing on a pool may block trying to create a new worker which uses GFP_KERNEL allocation which has slight chance of developing into deadlock if some works currently on the same queue need to be processed to satisfy the GFP_KERNEL allocation. This is the problem rescuer solves.

When such condition is possible, the pool summons rescuers of all workqueues which have works queued on the pool and let them process those works so that forward progress can be guaranteed.

This should happen rarely.

Return

0

void **check_flush_dependency**(struct workqueue_struct *target_wq, struct work_struct *target_work)

check for flush dependency sanity

Parameters

struct workqueue_struct *target_wq
workqueue being flushed

struct work_struct *target_work
work item being flushed (NULL for workqueue flushes)

Description

`current` is trying to flush the whole **target_wq** or **target_work** on it. If **target_wq** doesn't have `WQ_MEM_RECLAIM`, verify that `current` is not reclaiming memory or running on a workqueue which doesn't have `WQ_MEM_RECLAIM` as that can break forward-progress guarantee leading to a deadlock.

void **insert_wq_barrier**(struct pool_workqueue *pwq, struct wq_barrier *barr, struct work_struct *target, struct *worker* *worker)

insert a barrier work

Parameters

struct pool_workqueue *pwq
pwq to insert barrier into

struct wq_barrier *barr
wq_barrier to insert

struct work_struct *target
target work to attach **barr** to

struct worker *worker
worker currently executing **target**, NULL if **target** is not executing

Description

barr is linked to **target** such that **barr** is completed only after **target** finishes execution. Please note that the ordering guarantee is observed only with respect to **target** and on the local cpu.

Currently, a queued barrier can't be canceled. This is because *try_to_grab_pending()* can't determine whether the work to be grabbed is at the head of the queue and thus can't clear LINKED flag of the previous work while there must be a valid next work after a work with LINKED flag set.

Note that when **worker** is non-NULL, **target** may be modified underneath us, so we can't reliably determine pwq from **target**.

Context

raw_spin_lock_irq(pool->lock).

bool **flush_workqueue_prep_pwqs**(struct workqueue_struct *wq, int flush_color, int work_color)

prepare pwqs for workqueue flushing

Parameters

struct workqueue_struct *wq
workqueue being flushed

int flush_color
new flush color, < 0 for no-op

int work_color
new work color, < 0 for no-op

Description

Prepare pwqs for workqueue flushing.

If **flush_color** is non-negative, flush_color on all pwqs should be -1. If no pwq has in-flight commands at the specified color, all pwq->flush_color's stay at -1 and `false` is returned. If any pwq has in flight commands, its pwq->flush_color is set to **flush_color**, **wq->nr_pwqs_to_flush** is updated accordingly, pwq wakeup logic is armed and `true` is returned.

The caller should have initialized **wq->first_flusher** prior to calling this function with non-negative **flush_color**. If **flush_color** is negative, no flush color update is done and `false` is returned.

If **work_color** is non-negative, all pwqs should have the same work_color which is previous to **work_color** and all will be advanced to **work_color**.

Context

mutex_lock(wq->mutex).

Return

`true` if **flush_color** >= 0 and there's something to flush. `false` otherwise.

void **__flush_workqueue**(struct workqueue_struct *wq)

ensure that any scheduled work has run to completion.

Parameters

struct workqueue_struct *wq
workqueue to flush

Description

This function sleeps until all work items which were queued on entry have finished execution, but it is not livelocked by new incoming ones.

void **drain_workqueue**(struct workqueue_struct *wq)
drain a workqueue

Parameters

struct workqueue_struct *wq
workqueue to drain

Description

Wait until the workqueue becomes empty. While draining is in progress, only chain queueing is allowed. IOW, only currently pending or running work items on **wq** can queue further work items on it. **wq** is flushed repeatedly until it becomes empty. The number of flushing is determined by the depth of chaining and should be relatively short. Whine if it takes too long.

bool **flush_work**(struct work_struct *work)
wait for a work to finish executing the last queueing instance

Parameters

struct work_struct *work
the work to flush

Description

Wait until **work** has finished execution. **work** is guaranteed to be idle on return if it hasn't been requeued since flush started.

Return

`true` if *flush_work()* waited for the work to finish execution, `false` if it was already idle.

bool **cancel_work_sync**(struct work_struct *work)
cancel a work and wait for it to finish

Parameters

struct work_struct *work
the work to cancel

Description

Cancel **work** and wait for its execution to finish. This function can be used even if the work re-queues itself or migrates to another workqueue. On return from this function, **work** is guaranteed to be not pending or executing on any CPU.

cancel_work_sync(`delayed_work->work`) must not be used for delayed_work's. Use *cancel_delayed_work_sync()* instead.

The caller must ensure that the workqueue on which **work** was last queued can't be destroyed before this function returns.

Return

`true` if **work** was pending, `false` otherwise.

bool **flush_delayed_work**(struct delayed_work *dwork)
wait for a dwork to finish executing the last queueing

Parameters

struct delayed_work *dwork
the delayed work to flush

Description

Delayed timer is cancelled and the pending work is queued for immediate execution. Like *flush_work()*, this function only considers the last queueing instance of **dwork**.

Return

`true` if *flush_work()* waited for the work to finish execution, `false` if it was already idle.

bool **flush_rcu_work**(struct rcu_work *rwork)
wait for a rwork to finish executing the last queueing

Parameters

struct rcu_work *rwork
the rcu work to flush

Return

`true` if *flush_rcu_work()* waited for the work to finish execution, `false` if it was already idle.

bool **cancel_delayed_work**(struct delayed_work *dwork)
cancel a delayed work

Parameters

struct delayed_work *dwork
delayed_work to cancel

Description

Kill off a pending delayed_work.

This function is safe to call from any context including IRQ handler.

Return

`true` if **dwork** was pending and canceled; `false` if it wasn't pending.

Note

The work callback function may still be running on return, unless it returns `true` and the work doesn't re-arm itself. Explicitly flush or use *cancel_delayed_work_sync()* to wait on it.

bool **cancel_delayed_work_sync**(struct delayed_work *dwork)
cancel a delayed work and wait for it to finish

Parameters

struct delayed_work *dwork
the delayed work cancel

Description

This is *cancel_work_sync()* for delayed works.

Return

`true` if **dwork** was pending, `false` otherwise.

int **schedule_on_each_cpu**(work_func_t func)
execute a function synchronously on each online CPU

Parameters

work_func_t func
the function to call

Description

schedule_on_each_cpu() executes **func** on each online CPU using the system workqueue and blocks until all CPUs have completed. *schedule_on_each_cpu()* is very slow.

Return

0 on success, -errno on failure.

int **execute_in_process_context**(work_func_t fn, struct execute_work *ew)
reliably execute the routine with user context

Parameters

work_func_t fn
the function to execute

struct execute_work *ew
guaranteed storage for the execute work structure (must be available when the work executes)

Description

Executes the function immediately if process context is available, otherwise schedules the function for delayed execution.

Return

0 - function was executed
1 - function was scheduled for execution

void **free_workqueue_attrs**(struct *workqueue_attrs* *attrs)
free a workqueue_attrs

Parameters

struct workqueue_attrs *attrs
workqueue_attrs to free

Description

Undo *alloc_workqueue_attrs()*.

struct *workqueue_attrs* ***alloc_workqueue_attrs**(void)
allocate a workqueue_attrs

Parameters

void
no arguments

Description

Allocate a new workqueue_attrs, initialize with default settings and return it.

Return

The allocated new workqueue_attr on success. NULL on failure.

int **init_worker_pool**(struct worker_pool *pool)
initialize a newly zalloc'd worker_pool

Parameters

struct worker_pool *pool
worker_pool to initialize

Description

Initialize a newly zalloc'd **pool**. It also allocates **pool->attrs**.

Return

0 on success, -errno on failure. Even on failure, all fields inside **pool** proper are initialized and *put_unbound_pool()* can be called on **pool** safely to release it.

void **put_unbound_pool**(struct worker_pool *pool)
put a worker_pool

Parameters

struct worker_pool *pool
worker_pool to put

Description

Put **pool**. If its refcnt reaches zero, it gets destroyed in RCU safe manner. *get_unbound_pool()* calls this function on its failure path and this function should be able to release pools which went through, successfully or not, *init_worker_pool()*.

Should be called with wq_pool_mutex held.

struct worker_pool ***get_unbound_pool**(const struct *workqueue_attrs* *attrs)
get a worker_pool with the specified attributes

Parameters

const struct workqueue_attrs *attrs
the attributes of the worker_pool to get

Description

Obtain a worker_pool which has the same attributes as **attrs**, bump the reference count and return it. If there already is a matching worker_pool, it will be used; otherwise, this function attempts to create a new one.

Should be called with wq_pool_mutex held.

Return

On success, a worker_pool with the same attributes as **attrs**. On failure, NULL.

void **pwq_adjust_max_active**(struct pool_workqueue *pwq)
update a pwq's max_active to the current setting

Parameters

struct pool_workqueue *pwq
target pool_workqueue

Description

If **pwq** isn't freezing, set **pwq->max_active** to the associated workqueue's saved_max_active and activate inactive work items accordingly. If **pwq** is freezing, clear **pwq->max_active** to zero.

void **wq_calc_pod_cpumask**(struct *workqueue_attrs* *attrs, int cpu, int cpu_going_down)
calculate a wq_attrs' cpumask for a pod

Parameters

struct workqueue_attrs *attrs
the wq_attrs of the default pwq of the target workqueue

int cpu
the target CPU

int cpu_going_down
if >= 0, the CPU to consider as offline

Description

Calculate the cpumask a workqueue with **attrs** should use on **pod**. If **cpu_going_down** is >= 0, that cpu is considered offline during calculation. The result is stored in **attrs->__pod_cpumask**.

If pod affinity is not enabled, **attrs->cpumask** is always used. If enabled and **pod** has online CPUs requested by **attrs**, the returned cpumask is the intersection of the possible CPUs of **pod** and **attrs->cpumask**.

The caller is responsible for ensuring that the cpumask of **pod** stays stable.

int **apply_workqueue_attrs**(struct workqueue_struct *wq, const struct *workqueue_attrs* *attrs)
apply new workqueue_attrs to an unbound workqueue

Parameters

struct workqueue_struct *wq
the target workqueue

const struct workqueue_attrs *attrs
the workqueue_attrs to apply, allocated with *alloc_workqueue_attrs()*

Description

Apply **attrs** to an unbound workqueue **wq**. Unless disabled, this function maps a separate pwq to each CPU pod with possibles CPUs in **attrs->cpumask** so that work items are affine to the pod it was issued on. Older pwqs are released as in-flight work items finish. Note that a work item which repeatedly requeues itself back-to-back will stay on its current pwq.

Performs GFP_KERNEL allocations.

Assumes caller has CPU hotplug read exclusion, i.e. cpus_read_lock().

Return

0 on success and -errno on failure.

void **wq_update_pod**(struct workqueue_struct *wq, int cpu, int hotplug_cpu, bool online)
update pod affinity of a wq for CPU hot[un]plug

Parameters

struct workqueue_struct *wq
the target workqueue

int cpu
the CPU to update pool association for

int hotplug_cpu
the CPU coming up or going down

bool online
whether **cpu** is coming up or going down

Description

This function is to be called from `CPU_DOWN_PREPARE`, `CPU_ONLINE` and `CPU_DOWN_FAILED`. **cpu** is being hot[un]plugged, update pod affinity of **wq** accordingly.

If pod affinity can't be adjusted due to memory allocation failure, it falls back to **wq->dfl_pwq** which may not be optimal but is always correct.

Note that when the last allowed CPU of a pod goes offline for a workqueue with a cpumask spanning multiple pods, the workers which were already executing the work items for the workqueue will lose their CPU affinity and may execute on any CPU. This is similar to how per-cpu workqueues behave on CPU_DOWN. If a workqueue user wants strict affinity, it's the user's responsibility to flush the work item from CPU_DOWN_PREPARE.

void **destroy_workqueue**(struct workqueue_struct *wq)
safely terminate a workqueue

Parameters

struct workqueue_struct *wq
target workqueue

Description

Safely destroy a workqueue. All work currently pending will be done first.

void **workqueue_set_max_active**(struct workqueue_struct *wq, int max_active)
adjust max_active of a workqueue

Parameters

struct workqueue_struct *wq
target workqueue

int max_active
new max_active value.

Description

Set max_active of **wq** to **max_active**.

Context

Don't call from IRQ context.

struct work_struct ***current_work**(void)
retrieve `current` task's work struct

Parameters

void
no arguments

Description

Determine if `current` task is a workqueue worker and what it's working on. Useful to find out the context that the `current` task is running in.

Return

work struct if `current` task is a workqueue worker, `NULL` otherwise.

bool **current_is_workqueue_rescuer**(void)
is `current` workqueue rescuer?

Parameters

void
no arguments

Description

Determine whether `current` is a workqueue rescuer. Can be used from work functions to determine whether it's being run off the rescuer task.

Return

`true` if `current` is a workqueue rescuer. `false` otherwise.

bool **workqueue_congested**(int cpu, struct workqueue_struct *wq)
test whether a workqueue is congested

Parameters

int cpu
CPU in question

struct workqueue_struct *wq
target workqueue

Description

Test whether **wq**'s cpu workqueue for **cpu** is congested. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

If **cpu** is WORK_CPU_UNBOUND, the test is performed on the local CPU.

With the exception of ordered workqueues, all workqueues have per-cpu pool_workqueues, each with its own congested state. A workqueue being congested on one CPU doesn't mean that the workqueue is contested on any other CPUs.

Return

`true` if congested, `false` otherwise.

unsigned int **work_busy**(struct work_struct *work)

test whether a work is currently pending or running

Parameters

struct work_struct *work
the work to be tested

Description

Test whether **work** is currently pending or running. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

Return

OR'd bitmask of WORK_BUSY_* bits.

void **set_worker_desc**(const char *fmt, ...)

set description for the current work item

Parameters

const char *fmt
printf-style format string

...
arguments for the format string

Description

This function can be called by a running work function to describe what the work item is about. If the worker task gets dumped, this information will be printed out together to help debugging. The description can be at most WORKER_DESC_LEN including the trailing '0'.

void **print_worker_info**(const char *log_lvl, struct task_struct *task)

print out worker information and description

Parameters

const char *log_lvl
the log level to use when printing

struct task_struct *task
target task

Description

If **task** is a worker and currently executing a work item, print out the name of the workqueue being serviced and worker description set with *set_worker_desc()* by the currently executing work item.

This function can be safely called on any task as long as the task_struct itself is accessible. While safe, this function isn't synchronized and may print out mixups or garbages of limited length.

void **show_one_workqueue**(struct workqueue_struct *wq)

dump state of specified workqueue

Parameters

struct workqueue_struct *wq
workqueue whose state will be printed

void **show_one_worker_pool**(struct worker_pool *pool)
dump state of specified worker pool

Parameters

struct worker_pool *pool
worker pool whose state will be printed

void **show_all_workqueues**(void)
dump workqueue state

Parameters

void
no arguments

Description

Called from a sysrq handler and prints out all busy workqueues and pools.

void **show_freezable_workqueues**(void)
dump freezable workqueue state

Parameters

void
no arguments

Description

Called from try_to_freeze_tasks() and prints out all freezable workqueues still busy.

void **rebind_workers**(struct worker_pool *pool)
rebind all workers of a pool to the associated CPU

Parameters

struct worker_pool *pool
pool of interest

Description

pool->cpu is coming online. Rebind all workers to the CPU.

void **restore_unbound_workers_cpumask**(struct worker_pool *pool, int cpu)
restore cpumask of unbound workers

Parameters

struct worker_pool *pool
unbound pool of interest

int cpu
the CPU which is coming up

Description

An unbound pool may end up with a cpumask which doesn't have any online CPUs. When a worker of such pool get scheduled, the scheduler resets its cpus_allowed. If **cpu** is in **pool**'s cpumask which didn't have any online CPU before, cpus_allowed of all its workers should be restored.

long **work_on_cpu_key**(int cpu, long (*fn)(void*), void *arg, struct lock_class_key *key)
run a function in thread context on a particular cpu

Parameters

int cpu
the cpu to run on

long (*fn)(void *)
the function to run

void *arg
the function arg

struct lock_class_key *key
The lock class key for lock debugging purposes

Description

It is up to the caller to ensure that the cpu doesn't go offline. The caller must not hold any locks which would prevent **fn** from completing.

Return

The value **fn** returns.

long **work_on_cpu_safe_key**(int cpu, long (*fn)(void*), void *arg, struct lock_class_key *key)
run a function in thread context on a particular cpu

Parameters

int cpu
the cpu to run on

long (*fn)(void *)
the function to run

void *arg
the function argument

struct lock_class_key *key
The lock class key for lock debugging purposes

Description

Disables CPU hotplug and calls work_on_cpu(). The caller must not hold any locks which would prevent **fn** from completing.

Return

The value **fn** returns.

void **freeze_workqueues_begin**(void)
begin freezing workqueues

Parameters

void
: no arguments

Description

Start freezing workqueues. After this function returns, all freezable workqueues will queue new works to their inactive_works list instead of pool->worklist.

Context

Grabs and releases wq_pool_mutex, wq->mutex and pool->lock's.

bool **freeze_workqueues_busy**(void)
: are freezable workqueues still busy?

Parameters

void
: no arguments

Description

Check whether freezing is complete. This function must be called between *freeze_workqueues_begin()* and *thaw_workqueues()*.

Context

Grabs and releases wq_pool_mutex.

Return

`true` if some freezable workqueues are still busy. `false` if freezing is complete.

void **thaw_workqueues**(void)
: thaw workqueues

Parameters

void
: no arguments

Description

Thaw workqueues. Normal queueing is restored and all collected frozen works are transferred to their respective pool worklists.

Context

Grabs and releases wq_pool_mutex, wq->mutex and pool->lock's.

int **workqueue_set_unbound_cpumask**(cpumask_var_t cpumask)
: Set the low-level unbound cpumask

Parameters

cpumask_var_t cpumask
: the cpumask to set

The low-level workqueues cpumask is a global cpumask that limits the affinity of all unbound workqueues. This function check the **cpumask** and apply it to all unbound workqueues and updates all pwqs of them.

Return

0 - Success
-EINVAL - Invalid **cpumask** -ENOMEM - Failed to allocate memory for attrs or pwqs.

int **workqueue_sysfs_register**(struct workqueue_struct *wq)
make a workqueue visible in sysfs

Parameters

struct workqueue_struct *wq
the workqueue to register

Description

Expose **wq** in sysfs under /sys/bus/workqueue/devices. alloc_workqueue*() automatically calls this function if WQ_SYSFS is set which is the preferred method.

Workqueue user should use this function directly iff it wants to apply workqueue_attrs before making the workqueue visible in sysfs; otherwise, *apply_workqueue_attrs()* may race against userland updating the attributes.

Return

0 on success, -errno on failure.

void **workqueue_sysfs_unregister**(struct workqueue_struct *wq)
undo *workqueue_sysfs_register()*

Parameters

struct workqueue_struct *wq
the workqueue to unregister

Description

If **wq** is registered to sysfs by *workqueue_sysfs_register()*, unregister.

void **workqueue_init_early**(void)
early init for workqueue subsystem

Parameters

void
no arguments

Description

This is the first step of three-staged workqueue subsystem initialization and invoked as soon as the bare basics - memory allocation, cpumasks and idr are up. It sets up all the data structures and system workqueues and allows early boot code to create workqueues and queue/cancel work items. Actual work item execution starts only after kthreads can be created and scheduled right before early initcalls.

void **workqueue_init**(void)
bring workqueue subsystem fully online

Parameters

void
no arguments

Description

This is the second step of three-staged workqueue subsystem initialization and invoked as soon as kthreads can be created and scheduled. Workqueues have been created and work items queued on them, but there are no kworkers executing the work items yet. Populate the worker pools with the initial workers and enable future kworker creations.

void **workqueue_init_topology**(void)

initialize CPU pods for unbound workqueues

Parameters

void

no arguments

Description

This is the third step of there-staged workqueue subsystem initialization and invoked after SMP and topology information are fully initialized. It initializes the unbound CPU pods accordingly.

1.3 General notification mechanism

The general notification mechanism is built on top of the standard pipe driver whereby it effectively splices notification messages from the kernel into pipes opened by userspace. This can be used in conjunction with:

```
* Key/keyring notifications
```

The notifications buffers can be enabled by:

“General setup”/”General notification queue” (CONFIG_WATCH_QUEUE)

This document has the following sections:

1.3.1 Overview

This facility appears as a pipe that is opened in a special mode. The pipe's internal ring buffer is used to hold messages that are generated by the kernel. These messages are then read out by read(). Splice and similar are disabled on such pipes due to them wanting to, under some circumstances, revert their additions to the ring - which might end up interleaved with notification messages.

The owner of the pipe has to tell the kernel which sources it would like to watch through that pipe. Only sources that have been connected to a pipe will insert messages into it. Note that a source may be bound to multiple pipes and insert messages into all of them simultaneously.

Filters may also be emplaced on a pipe so that certain source types and subevents can be ignored if they're not of interest.

A message will be discarded if there isn't a slot available in the ring or if no pre-allocated message buffer is available. In both of these cases, read() will insert a WATCH_META_LOSS_NOTIFICATION message into the output buffer after the last message currently in the buffer has been read.

Note that when producing a notification, the kernel does not wait for the consumers to collect it, but rather just continues on. This means that notifications can be generated whilst spinlocks are held and also protects the kernel from being held up indefinitely by a userspace malfunction.

1.3.2 Message Structure

Notification messages begin with a short header:

```
struct watch_notification {
        __u32   type:24;
        __u32   subtype:8;
        __u32   info;
};
```

"type" indicates the source of the notification record and "subtype" indicates the type of record from that source (see the Watch Sources section below). The type may also be "WATCH_TYPE_META". This is a special record type generated internally by the watch queue itself. There are two subtypes:

- WATCH_META_REMOVAL_NOTIFICATION
- WATCH_META_LOSS_NOTIFICATION

The first indicates that an object on which a watch was installed was removed or destroyed and the second indicates that some messages have been lost.

"info" indicates a bunch of things, including:

- The length of the message in bytes, including the header (mask with WATCH_INFO_LENGTH and shift by WATCH_INFO_LENGTH__SHIFT). This indicates the size of the record, which may be between 8 and 127 bytes.
- The watch ID (mask with WATCH_INFO_ID and shift by WATCH_INFO_ID__SHIFT). This indicates that caller's ID of the watch, which may be between 0 and 255. Multiple watches may share a queue, and this provides a means to distinguish them.

- A type-specific field (WATCH_INFO_TYPE_INFO). This is set by the notification producer to indicate some meaning specific to the type and subtype.

Everything in info apart from the length can be used for filtering.

The header can be followed by supplementary information. The format of this is at the discretion is defined by the type and subtype.

1.3.3 Watch List (Notification Source) API

A "watch list" is a list of watchers that are subscribed to a source of notifications. A list may be attached to an object (say a key or a superblock) or may be global (say for device events). From a userspace perspective, a non-global watch list is typically referred to by reference to the object it belongs to (such as using KEYCTL_NOTIFY and giving it a key serial number to watch that specific key).

To manage a watch list, the following functions are provided:

- ```
  void init_watch_list(struct watch_list *wlist,
                       void (*release_watch)(struct watch *wlist));
  ```

  Initialise a watch list. If `release_watch` is not NULL, then this indicates a function that should be called when the watch_list object is destroyed to discard any references the watch list holds on the watched object.

- `void remove_watch_list(struct watch_list *wlist);`

  This removes all of the watches subscribed to a watch_list and frees them and then destroys the watch_list object itself.

1.3.4 Watch Queue (Notification Output) API

A "watch queue" is the buffer allocated by an application that notification records will be written into. The workings of this are hidden entirely inside of the pipe device driver, but it is necessary to gain a reference to it to set a watch. These can be managed with:

- `struct watch_queue *get_watch_queue(int fd);`

  Since watch queues are indicated to the kernel by the fd of the pipe that implements the buffer, userspace must hand that fd through a system call. This can be used to look up an opaque pointer to the watch queue from the system call.

- `void put_watch_queue(struct watch_queue *wqueue);`

  This discards the reference obtained from `get_watch_queue()`.
  ```

1.3.5 Watch Subscription API

A "watch" is a subscription on a watch list, indicating the watch queue, and thus the buffer, into which notification records should be written. The watch queue object may also carry filtering rules for that object, as set by userspace. Some parts of the watch struct can be set by the driver:

```
struct watch {
        union {
                u32             info_id;        /* ID to be OR'd in to info␣
↪field */
                ...
        };
        void                    *private;       /* Private data for the␣
↪watched object */
        u64                     id;             /* Internal identifier */
        ...
};
```

The info_id value should be an 8-bit number obtained from userspace and shifted by WATCH_INFO_ID__SHIFT. This is OR'd into the WATCH_INFO_ID field of struct watch_notification::info when and if the notification is written into the associated watch queue buffer.

The private field is the driver's data associated with the watch_list and is cleaned up by the watch_list::release_watch() method.

The id field is the source's ID. Notifications that are posted with a different ID are ignored.

The following functions are provided to manage watches:

- `void init_watch(struct watch *watch, struct watch_queue *wqueue);`

  Initialise a watch object, setting its pointer to the watch queue, using appropriate barriering to avoid lockdep complaints.

- `int add_watch_to_object(struct watch *watch, struct watch_list *wlist);`

  Subscribe a watch to a watch list (notification source). The driver-settable fields in the watch struct must have been set before this is called.

- ```
  int remove_watch_from_object(struct watch_list *wlist,
                               struct watch_queue *wqueue,
                               u64 id, false);
  ```

  Remove a watch from a watch list, where the watch must match the specified watch queue (wqueue) and object identifier (id). A notification (WATCH_META_REMOVAL_NOTIFICATION) is sent to the watch queue to indicate that the watch got removed.

- `int remove_watch_from_object(struct watch_list *wlist, NULL, 0, true);`

  Remove all the watches from a watch list. It is expected that this will be called preparatory to destruction and that the watch list will be inaccessible to new watches by this point. A notification (WATCH_META_REMOVAL_NOTIFICATION) is sent to the watch queue of each subscribed watch to indicate that the watch got removed.
  ```

1.3.6 Notification Posting API

To post a notification to watch list so that the subscribed watches can see it, the following function should be used:

```
void post_watch_notification(struct watch_list *wlist,
                             struct watch_notification *n,
                             const struct cred *cred,
                             u64 id);
```

The notification should be preformatted and a pointer to the header (`n`) should be passed in. The notification may be larger than this and the size in units of buffer slots is noted in `n->info & WATCH_INFO_LENGTH`.

The `cred` struct indicates the credentials of the source (subject) and is passed to the LSMs, such as SELinux, to allow or suppress the recording of the note in each individual queue according to the credentials of that queue (object).

The `id` is the ID of the source object (such as the serial number on a key). Only watches that have the same ID set in them will see this notification.

1.3.7 Watch Sources

Any particular buffer can be fed from multiple sources. Sources include:

- WATCH_TYPE_KEY_NOTIFY

  Notifications of this type indicate changes to keys and keyrings, including the changes of keyring contents or the attributes of keys.

  See Documentation/security/keys/core.rst for more information.

1.3.8 Event Filtering

Once a watch queue has been created, a set of filters can be applied to limit the events that are received using:

```
struct watch_notification_filter filter = {
        ...
};
ioctl(fd, IOC_WATCH_QUEUE_SET_FILTER, &filter)
```

The filter description is a variable of type:

```
struct watch_notification_filter {
        __u32   nr_filters;
        __u32   __reserved;
        struct watch_notification_type_filter filters[];
};
```

Where "nr_filters" is the number of filters in filters[] and "__reserved" should be 0. The "filters" array has elements of the following type:

```
struct watch_notification_type_filter {
        __u32   type;
        __u32   info_filter;
        __u32   info_mask;
        __u32   subtype_filter[8];
};
```

Where:

- `type` is the event type to filter for and should be something like "WATCH_TYPE_KEY_NOTIFY"
- `info_filter` and `info_mask` act as a filter on the info field of the notification record. The notification is only written into the buffer if:

  ```
  (watch.info & info_mask) == info_filter
  ```

  This could be used, for example, to ignore events that are not exactly on the watched point in a mount tree.
- `subtype_filter` is a bitmask indicating the subtypes that are of interest. Bit 0 of subtype_filter[0] corresponds to subtype 0, bit 1 to subtype 1, and so on.

If the argument to the ioctl() is NULL, then the filters will be removed and all events from the watched sources will come through.

1.3.9 Userspace Code Example

A buffer is created with something like the following:

```
pipe2(fds, O_TMPFILE);
ioctl(fds[1], IOC_WATCH_QUEUE_SET_SIZE, 256);
```

It can then be set to receive keyring change notifications:

```
keyctl(KEYCTL_WATCH_KEY, KEY_SPEC_SESSION_KEYRING, fds[1], 0x01);
```

The notifications can then be consumed by something like the following:

```
static void consumer(int rfd, struct watch_queue_buffer *buf)
{
        unsigned char buffer[128];
        ssize_t buf_len;

        while (buf_len = read(rfd, buffer, sizeof(buffer)),
               buf_len > 0
               ) {
                void *p = buffer;
                void *end = buffer + buf_len;
                while (p < end) {
                        union {
                                struct watch_notification n;
                                unsigned char buf1[128];
```

```
                        } n;
                        size_t largest, len;

                        largest = end - p;
                        if (largest > 128)
                                largest = 128;
                        memcpy(&n, p, largest);

                        len = (n->info & WATCH_INFO_LENGTH) >>
                                WATCH_INFO_LENGTH__SHIFT;
                        if (len == 0 || len > largest)
                                return;

                        switch (n.n.type) {
                        case WATCH_TYPE_META:
                                got_meta(&n.n);
                        case WATCH_TYPE_KEY_NOTIFY:
                                saw_key_change(&n.n);
                                break;
                        }

                        p += len;
                }
        }
}
```

1.4 Message logging with printk

printk() is one of the most widely known functions in the Linux kernel. It's the standard tool we have for printing messages and usually the most basic way of tracing and debugging. If you're familiar with printf(3) you can tell printk() is based on it, although it has some functional differences:

- printk() messages can specify a log level.
- the format string, while largely compatible with C99, doesn't follow the exact same specification. It has some extensions and a few limitations (no `%n` or floating point conversion specifiers). See *How to get printk format specifiers right*.

All printk() messages are printed to the kernel log buffer, which is a ring buffer exported to userspace through /dev/kmsg. The usual way to read it is using `dmesg`.

printk() is typically used like this:

```
printk(KERN_INFO "Message: %s\n", arg);
```

where `KERN_INFO` is the log level (note that it's concatenated to the format string, the log level is not a separate argument). The available log levels are:

Name	String	Alias function
KERN_EMERG	"0"	pr_emerg()
KERN_ALERT	"1"	pr_alert()
KERN_CRIT	"2"	pr_crit()
KERN_ERR	"3"	pr_err()
KERN_WARNING	"4"	pr_warn()
KERN_NOTICE	"5"	pr_notice()
KERN_INFO	"6"	pr_info()
KERN_DEBUG	"7"	pr_debug() and pr_devel() if DEBUG is defined
KERN_DEFAULT	""	
KERN_CONT	"c"	pr_cont()

The log level specifies the importance of a message. The kernel decides whether to show the message immediately (printing it to the current console) depending on its log level and the current *console_loglevel* (a kernel variable). If the message priority is higher (lower log level value) than the *console_loglevel* the message will be printed to the console.

If the log level is omitted, the message is printed with `KERN_DEFAULT` level.

You can check the current *console_loglevel* with:

```
$ cat /proc/sys/kernel/printk
4        4        1        7
```

The result shows the *current*, *default*, *minimum* and *boot-time-default* log levels.

To change the current console_loglevel simply write the desired level to `/proc/sys/kernel/printk`. For example, to print all messages to the console:

```
# echo 8 > /proc/sys/kernel/printk
```

Another way, using `dmesg`:

```
# dmesg -n 5
```

sets the console_loglevel to print KERN_WARNING (4) or more severe messages to console. See `dmesg(1)` for more information.

As an alternative to printk() you can use the `pr_*()` aliases for logging. This family of macros embed the log level in the macro names. For example:

```
pr_info("Info message no. %d\n", msg_num);
```

prints a `KERN_INFO` message.

Besides being more concise than the equivalent printk() calls, they can use a common definition for the format string through the pr_fmt() macro. For instance, defining this at the top of a source file (before any `#include` directive):

```
#define pr_fmt(fmt) "%s:%s: " fmt, KBUILD_MODNAME, __func__
```

would prefix every pr_*() message in that file with the module and function name that originated the message.

For debugging purposes there are also two conditionally-compiled macros: pr_debug() and pr_devel(), which are compiled-out unless `DEBUG` (or also `CONFIG_DYNAMIC_DEBUG` in the case of pr_debug()) is defined.

1.4.1 Function reference

pr_fmt

`pr_fmt (fmt)`

used by the pr_*() macros to generate the printk format string

Parameters

fmt
format string passed from a pr_*() macro

Description

This macro can be used to generate a unified format string for pr_*() macros. A common use is to prefix all pr_*() messages in a file with a common string. For example, defining this at the top of a source file:

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

would prefix all pr_info, pr_emerg… messages in the file with the module name.

printk

`printk (fmt, ...)`

print a kernel message

Parameters

fmt
format string

...
variable arguments

Description

This is printk(). It can be called from any context. We want it to work.

If printk indexing is enabled, _printk() is called from printk_index_wrap. Otherwise, printk is simply #defined to _printk.

We try to grab the console_lock. If we succeed, it's easy - we log the output and call the console drivers. If we fail to get the semaphore, we place the output into the log buffer and return. The current holder of the console_sem will notice the new output in console_unlock(); and will send it to the consoles before releasing the lock.

One effect of this deferred printing is that code which calls printk() and then changes console_loglevel may break. This is because console_loglevel is inspected when the actual printing occurs.

See also: printf(3)

See the vsnprintf() documentation for format string extensions over C99.

pr_emerg

`pr_emerg (fmt, ...)`

Print an emergency-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_EMERG loglevel. It uses pr_fmt() to generate the format string.

pr_alert

`pr_alert (fmt, ...)`

Print an alert-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_ALERT loglevel. It uses pr_fmt() to generate the format string.

pr_crit

`pr_crit (fmt, ...)`

Print a critical-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_CRIT loglevel. It uses pr_fmt() to generate the format string.

pr_err

`pr_err (fmt, ...)`

Print an error-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_ERR loglevel. It uses pr_fmt() to generate the format string.

pr_warn

`pr_warn (fmt, ...)`

Print a warning-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_WARNING loglevel. It uses pr_fmt() to generate the format string.

pr_notice

`pr_notice (fmt, ...)`

Print a notice-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_NOTICE loglevel. It uses pr_fmt() to generate the format string.

pr_info

`pr_info (fmt, ...)`

Print an info-level message

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_INFO loglevel. It uses pr_fmt() to generate the format string.

pr_cont

`pr_cont (fmt, ...)`

Continues a previous log message in the same line.

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_CONT loglevel. It should only be used when continuing a log message with no newline ('n') enclosed. Otherwise it defaults back to KERN_DEFAULT loglevel.

pr_devel

`pr_devel (fmt, ...)`

Print a debug-level message conditionally

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to a printk with KERN_DEBUG loglevel if DEBUG is defined. Otherwise it does nothing.

It uses pr_fmt() to generate the format string.

pr_debug

`pr_debug (fmt, ...)`

Print a debug-level message conditionally

Parameters

fmt
format string

...
arguments for the format string

Description

This macro expands to dynamic_pr_debug() if CONFIG_DYNAMIC_DEBUG is set. Otherwise, if DEBUG is defined, it's equivalent to a printk with KERN_DEBUG loglevel. If DEBUG is not defined it does nothing.

It uses pr_fmt() to generate the format string (dynamic_pr_debug() uses pr_fmt() internally).

1.5 How to get printk format specifiers right

Author
Randy Dunlap <rdunlap@infradead.org>

Author
Andrew Murray <amurray@mpc-data.co.uk>

1.5.1 Integer types

```
If variable is of Type,         use printk format specifier:
------------------------------------------------------------
        signed char             %d or %hhx
        unsigned char           %u or %x
        char                    %u or %x
        short int               %d or %hx
        unsigned short int      %u or %x
        int                     %d or %x
        unsigned int            %u or %x
        long                    %ld or %lx
        unsigned long           %lu or %lx
        long long               %lld or %llx
        unsigned long long      %llu or %llx
        size_t                  %zu or %zx
        ssize_t                 %zd or %zx
        s8                      %d or %hhx
        u8                      %u or %x
        s16                     %d or %hx
        u16                     %u or %x
        s32                     %d or %x
        u32                     %u or %x
        s64                     %lld or %llx
        u64                     %llu or %llx
```

If <type> is architecture-dependent for its size (e.g., cycles_t, tcflag_t) or is dependent on a config option for its size (e.g., blk_status_t), use a format specifier of its largest possible type and explicitly cast to it.

Example:

```
printk("test: latency: %llu cycles\n", (unsigned long long)time);
```

Reminder: sizeof() returns type size_t.

The kernel's printf does not support %n. Floating point formats (%e, %f, %g, %a) are also not recognized, for obvious reasons. Use of any unsupported specifier or length qualifier results in a WARN and early return from vsnprintf().

1.5.2 Pointer types

A raw pointer value may be printed with %p which will hash the address before printing. The kernel also supports extended specifiers for printing pointers of different types.

Some of the extended specifiers print the data on the given address instead of printing the address itself. In this case, the following error messages might be printed instead of the unreachable information:

```
(null)   data on plain NULL address
(efault) data on invalid address
(einval) invalid data on a valid address
```

Plain Pointers

```
%p      abcdef12 or 00000000abcdef12
```

Pointers printed without a specifier extension (i.e unadorned %p) are hashed to prevent leaking information about the kernel memory layout. This has the added benefit of providing a unique identifier. On 64-bit machines the first 32 bits are zeroed. The kernel will print `(ptrval)` until it gathers enough entropy.

When possible, use specialised modifiers such as %pS or %pB (described below) to avoid the need of providing an unhashed address that has to be interpreted post-hoc. If not possible, and the aim of printing the address is to provide more information for debugging, use %p and boot the kernel with the `no_hash_pointers` parameter during debugging, which will print all %p addresses unmodified. If you *really* always want the unmodified address, see %px below.

If (and only if) you are printing addresses as a content of a virtual file in e.g. procfs or sysfs (using e.g. seq_printf(), not printk()) read by a userspace process, use the %pK modifier described below instead of %p or %px.

Error Pointers

```
%pe     -ENOSPC
```

For printing error pointers (i.e. a pointer for which IS_ERR() is true) as a symbolic error name. Error values for which no symbolic name is known are printed in decimal, while a non-ERR_PTR passed as the argument to %pe gets treated as ordinary %p.

Symbols/Function Pointers

```
%pS     versatile_init+0x0/0x110
%ps     versatile_init
%pSR    versatile_init+0x9/0x110
        (with __builtin_extract_return_addr() translation)
%pB     prev_fn_of_versatile_init+0x88/0x88
```

The S and s specifiers are used for printing a pointer in symbolic format. They result in the symbol name with (S) or without (s) offsets. If KALLSYMS are disabled then the symbol address is printed instead.

The B specifier results in the symbol name with offsets and should be used when printing stack backtraces. The specifier takes into consideration the effect of compiler optimisations which may occur when tail-calls are used and marked with the noreturn GCC attribute.

If the pointer is within a module, the module name and optionally build ID is printed after the symbol name with an extra b appended to the end of the specifier.

```
%pS     versatile_init+0x0/0x110 [module_name]
%pSb    versatile_init+0x0/0x110 [module_name␣
↪ed5019fdf5e53be37cb1ba7899292d7e143b259e]
%pSRb   versatile_init+0x9/0x110 [module_name␣
↪ed5019fdf5e53be37cb1ba7899292d7e143b259e]
        (with __builtin_extract_return_addr() translation)
%pBb    prev_fn_of_versatile_init+0x88/0x88 [module_name␣
↪ed5019fdf5e53be37cb1ba7899292d7e143b259e]
```

Probed Pointers from BPF / tracing

```
%pks    kernel string
%pus    user string
```

The k and u specifiers are used for printing prior probed memory from either kernel memory (k) or user memory (u). The subsequent s specifier results in printing a string. For direct use in regular vsnprintf() the (k) and (u) annotation is ignored, however, when used out of BPF's bpf_trace_printk(), for example, it reads the memory it is pointing to without faulting.

Kernel Pointers

```
%pK     01234567 or 0123456789abcdef
```

For printing kernel pointers which should be hidden from unprivileged users. The behaviour of %pK depends on the kptr_restrict sysctl - see Documentation/admin-guide/sysctl/kernel.rst for more details.

This modifier is *only* intended when producing content of a file read by userspace from e.g. procfs or sysfs, not for dmesg. Please refer to the section about %p above for discussion about how to manage hashing pointers in printk().

Unmodified Addresses

```
%px     01234567 or 0123456789abcdef
```

For printing pointers when you *really* want to print the address. Please consider whether or not you are leaking sensitive information about the kernel memory layout before printing pointers with %px. %px is functionally equivalent to %lx (or %lu). %px is preferred because it is more uniquely grep'able. If in the future we need to modify the way the kernel handles printing pointers we will be better equipped to find the call sites.

Before using %px, consider if using %p is sufficient together with enabling the `no_hash_pointers` kernel parameter during debugging sessions (see the %p description above). One valid scenario for %px might be printing information immediately before a panic, which prevents any sensitive information to be exploited anyway, and with %px there would be no need to reproduce the panic with no_hash_pointers.

Pointer Differences

```
%td     2560
%tx     a00
```

For printing the pointer differences, use the %t modifier for ptrdiff_t.

Example:

```
printk("test: difference between pointers: %td\n", ptr2 - ptr1);
```

Struct Resources

```
%pr     [mem 0x60000000-0x6fffffff flags 0x2200] or
        [mem 0x0000000060000000-0x000000006fffffff flags 0x2200]
%pR     [mem 0x60000000-0x6fffffff pref] or
        [mem 0x0000000060000000-0x000000006fffffff pref]
```

For printing struct resources. The `R` and `r` specifiers result in a printed resource with (R) or without (r) a decoded flags member.

Passed by reference.

Physical address types phys_addr_t

```
%pa[p]  0x01234567 or 0x0123456789abcdef
```

For printing a phys_addr_t type (and its derivatives, such as resource_size_t) which can vary based on build options, regardless of the width of the CPU data path.

Passed by reference.

DMA address types dma_addr_t

```
%pad    0x01234567 or 0x0123456789abcdef
```

For printing a dma_addr_t type which can vary based on build options, regardless of the width of the CPU data path.

Passed by reference.

Raw buffer as an escaped string

```
%*pE[achnops]
```

For printing raw buffer as an escaped string. For the following buffer:

```
1b 62 20 5c 43 07 22 90 0d 5d
```

A few examples show how the conversion would be done (excluding surrounding quotes):

```
%*pE            "\eb \C\a"\220\r]"
%*pEhp          "\x1bb \C\x07"\x90\x0d]"
%*pEa           "\e\142\040\\\103\a\042\220\r\135"
```

The conversion rules are applied according to an optional combination of flags (see *string_escape_mem()* kernel documentation for the details):

- a - ESCAPE_ANY
- c - ESCAPE_SPECIAL
- h - ESCAPE_HEX
- n - ESCAPE_NULL
- o - ESCAPE_OCTAL
- p - ESCAPE_NP
- s - ESCAPE_SPACE

By default ESCAPE_ANY_NP is used.

ESCAPE_ANY_NP is the sane choice for many cases, in particularly for printing SSIDs.

If field width is omitted then 1 byte only will be escaped.

Raw buffer as a hex string

```
%*ph    00 01 02  ...  3f
%*phC   00:01:02: ... :3f
%*phD   00-01-02- ... -3f
%*phN   000102 ... 3f
```

For printing small buffers (up to 64 bytes long) as a hex string with a certain separator. For larger buffers consider using `print_hex_dump()`.

MAC/FDDI addresses

```
%pM     00:01:02:03:04:05
%pMR    05:04:03:02:01:00
%pMF    00-01-02-03-04-05
%pm     000102030405
%pmR    050403020100
```

For printing 6-byte MAC/FDDI addresses in hex notation. The `M` and `m` specifiers result in a printed address with (M) or without (m) byte separators. The default byte separator is the colon (:).

Where FDDI addresses are concerned the `F` specifier can be used after the `M` specifier to use dash (-) separators instead of the default separator.

For Bluetooth addresses the `R` specifier shall be used after the `M` specifier to use reversed byte order suitable for visual interpretation of Bluetooth addresses which are in the little endian order.

Passed by reference.

IPv4 addresses

```
%pI4    1.2.3.4
%pi4    001.002.003.004
%p[Ii]4[hnbl]
```

For printing IPv4 dot-separated decimal addresses. The `I4` and `i4` specifiers result in a printed address with (i4) or without (I4) leading zeros.

The additional `h`, `n`, `b`, and `l` specifiers are used to specify host, network, big or little endian order addresses respectively. Where no specifier is provided the default network/big endian order is used.

Passed by reference.

IPv6 addresses

```
%pI6    0001:0002:0003:0004:0005:0006:0007:0008
%pi6    00010002000300040005000600070008
%pI6c   1:2:3:4:5:6:7:8
```

For printing IPv6 network-order 16-bit hex addresses. The `I6` and `i6` specifiers result in a printed address with (I6) or without (i6) colon-separators. Leading zeros are always used.

The additional `c` specifier can be used with the `I` specifier to print a compressed IPv6 address as described by https://tools.ietf.org/html/rfc5952

Passed by reference.

IPv4/IPv6 addresses (generic, with port, flowinfo, scope)

```
%pIS    1.2.3.4         or 0001:0002:0003:0004:0005:0006:0007:0008
%piS    001.002.003.004 or 00010002000300040005000600070008
%pISc   1.2.3.4         or 1:2:3:4:5:6:7:8
%pISpc  1.2.3.4:12345   or [1:2:3:4:5:6:7:8]:12345
%p[Ii]S[pfschnbl]
```

For printing an IP address without the need to distinguish whether it's of type AF_INET or AF_INET6. A pointer to a valid struct sockaddr, specified through `IS` or `iS`, can be passed to this format specifier.

The additional `p`, `f`, and `s` specifiers are used to specify port (IPv4, IPv6), flowinfo (IPv6) and scope (IPv6). Ports have a `:` prefix, flowinfo a `/` and scope a `%`, each followed by the actual value.

In case of an IPv6 address the compressed IPv6 address as described by https://tools.ietf.org/html/rfc5952 is being used if the additional specifier `c` is given. The IPv6 address is surrounded by `[`, `]` in case of additional specifiers `p`, `f` or `s` as suggested by https://tools.ietf.org/html/draft-ietf-6man-text-addr-representation-07

In case of IPv4 addresses, the additional `h`, `n`, `b`, and `l` specifiers can be used as well and are ignored in case of an IPv6 address.

Passed by reference.

Further examples:

```
%pISfc          1.2.3.4         or [1:2:3:4:5:6:7:8]/123456789
%pISsc          1.2.3.4         or [1:2:3:4:5:6:7:8]%1234567890
%pISpfc         1.2.3.4:12345   or [1:2:3:4:5:6:7:8]:12345/123456789
```

UUID/GUID addresses

```
%pUb    00010203-0405-0607-0809-0a0b0c0d0e0f
%pUB    00010203-0405-0607-0809-0A0B0C0D0E0F
%pUl    03020100-0504-0706-0809-0a0b0c0e0e0f
%pUL    03020100-0504-0706-0809-0A0B0C0E0E0F
```

For printing 16-byte UUID/GUIDs addresses. The additional `l`, `L`, `b` and `B` specifiers are used to specify a little endian order in lower (l) or upper case (L) hex notation - and big endian order in lower (b) or upper case (B) hex notation.

Where no additional specifiers are used the default big endian order with lower case hex notation will be printed.

Passed by reference.

dentry names

```
%pd{,2,3,4}
%pD{,2,3,4}
```

For printing dentry name; if we race with `d_move()`, the name might be a mix of old and new ones, but it won't oops. %pd dentry is a safer equivalent of %s dentry->d_name.name we used to use, %pd<n> prints `n` last components. %pD does the same thing for struct file.

Passed by reference.

block_device names

```
%pg     sda, sda1 or loop0p1
```

For printing name of block_device pointers.

struct va_format

```
%pV
```

For printing struct va_format structures. These contain a format string and va_list as follows:

```
struct va_format {
        const char *fmt;
        va_list *va;
};
```

Implements a "recursive vsnprintf".

Do not use this feature without some mechanism to verify the correctness of the format string and va_list arguments.

Passed by reference.

Device tree nodes

```
%pOF[fnpPcCF]
```

For printing device tree node structures. Default behaviour is equivalent to %pOFf.

- f - device node full_name
- n - device node name
- p - device node phandle
- P - device node path spec (name + @unit)
- F - device node flags
- c - major compatible string
- C - full compatible string

The separator when using multiple arguments is ':'

Examples:

```
%pOF    /foo/bar@0                      - Node full name
%pOFf   /foo/bar@0                      - Same as above
%pOFfp  /foo/bar@0:10                   - Node full name + phandle
%pOFfcF /foo/bar@0:foo,device:--P-      - Node full name +
                                          major compatible string +
                                          node flags
                                                D - dynamic
                                                d - detached
                                                P - Populated
                                                B - Populated bus
```

Passed by reference.

Fwnode handles

```
%pfw[fP]
```

For printing information on fwnode handles. The default is to print the full node name, including the path. The modifiers are functionally equivalent to %pOF above.

- f - full name of the node, including the path
- P - the name of the node including an address (if there is one)

Examples (ACPI):

```
%pfwf   \_SB.PCI0.CIO2.port@1.endpoint@0        - Full node name
%pfwP   endpoint@0                              - Node name
```

Examples (OF):

```
%pfwf   /ocp@68000000/i2c@48072000/camera@10/port/endpoint - Full name
%pfwP   endpoint                                - Node name
```

Time and date

```
%pt[RT]                 YYYY-mm-ddTHH:MM:SS
%pt[RT]s                YYYY-mm-dd HH:MM:SS
%pt[RT]d                YYYY-mm-dd
%pt[RT]t                HH:MM:SS
%pt[RT][dt][r][s]
```

For printing date and time as represented by:

```
R  struct rtc_time structure
T  time64_t type
```

in human readable format.

By default year will be incremented by 1900 and month by 1. Use %pt[RT]r (raw) to suppress this behaviour.

The %pt[RT]s (space) will override ISO 8601 separator by using ' ' (space) instead of 'T' (Capital T) between date and time. It won't have any effect when date or time is omitted.

Passed by reference.

struct clk

```
%pC     pll1
%pCn    pll1
```

For printing struct clk structures. %pC and %pCn print the name of the clock (Common Clock Framework) or a unique 32-bit ID (legacy clock framework).

Passed by reference.

bitmap and its derivatives such as cpumask and nodemask

```
%*pb    0779
%*pbl   0,3-6,8-10
```

For printing bitmap and its derivatives such as cpumask and nodemask, %*pb outputs the bitmap with field width as the number of bits and %*pbl output the bitmap as range list with field width as the number of bits.

The field width is passed by value, the bitmap is passed by reference. Helper macros cpumask_pr_args() and nodemask_pr_args() are available to ease printing cpumask and nodemask.

Flags bitfields such as page flags, page_type, gfp_flags

```
%pGp    ␣
↪0x17ffffc0002036(referenced|uptodate|lru|active|private|node=0|zone=2|lastcpupid=0x
%pGt    0xffffff7f(buddy)
%pGg    GFP_USER|GFP_DMA32|GFP_NOWARN
%pGv    read|exec|mayread|maywrite|mayexec|denywrite
```

For printing flags bitfields as a collection of symbolic constants that would construct the value. The type of flags is given by the third character. Currently supported are:

- p - [p]age flags, expects value of type (`unsigned long *`)
- t - page [t]ype, expects value of type (`unsigned int *`)
- v - [v]ma_flags, expects value of type (`unsigned long *`)
- g - [g]fp_flags, expects value of type (`gfp_t *`)

The flag names and print order depends on the particular type.

Note that this format should not be used directly in the `TP_printk()` part of a tracepoint. Instead, use the show_*_flags() functions from <trace/events/mmflags.h>.

Passed by reference.

Network device features

```
%pNF    0x000000000000c000
```

For printing netdev_features_t.

Passed by reference.

V4L2 and DRM FourCC code (pixel format)

```
%p4cc
```

Print a FourCC code used by V4L2 or DRM, including format endianness and its numerical value as hexadecimal.

Passed by reference.

Examples:

```
%p4cc   BG12 little-endian (0x32314742)
%p4cc   Y10  little-endian (0x20303159)
%p4cc   NV12 big-endian (0xb231564e)
```

Rust

```
%pA
```

Only intended to be used from Rust code to format `core::fmt::Arguments`. Do *not* use it from C.

1.5.3 Thanks

If you add other %p extensions, please extend <lib/test_printf.c> with one or more test cases, if at all feasible.

Thank you for your cooperation and attention.

1.6 Printk Index

There are many ways how to monitor the state of the system. One important source of information is the system log. It provides a lot of information, including more or less important warnings and error messages.

There are monitoring tools that filter and take action based on messages logged.

The kernel messages are evolving together with the code. As a result, particular kernel messages are not KABI and never will be!

It is a huge challenge for maintaining the system log monitors. It requires knowing what messages were updated in a particular kernel version and why. Finding these changes in the sources would require non-trivial parsers. Also it would require matching the sources with the binary kernel which is not always trivial. Various changes might be backported. Various kernel versions might be used on different monitored systems.

This is where the printk index feature might become useful. It provides a dump of printk formats used all over the source code used for the kernel and modules on the running system. It is accessible at runtime via debugfs.

The printk index helps to find changes in the message formats. Also it helps to track the strings back to the kernel sources and the related commit.

1.6.1 User Interface

The index of printk formats are split in into separate files. The files are named according to the binaries where the printk formats are built-in. There is always "vmlinux" and optionally also modules, for example:

```
/sys/kernel/debug/printk/index/vmlinux
/sys/kernel/debug/printk/index/ext4
/sys/kernel/debug/printk/index/scsi_mod
```

Note that only loaded modules are shown. Also printk formats from a module might appear in "vmlinux" when the module is built-in.

The content is inspired by the dynamic debug interface and looks like:

```
$> head -1 /sys/kernel/debug/printk/index/vmlinux; shuf -n 5 vmlinux
# <level[,flags]> filename:line function "format"
<5> block/blk-settings.c:661 disk_stack_limits "%s: Warning: Device %s is␣
↪misaligned\n"
<4> kernel/trace/trace.c:8296 trace_create_file "Could not create tracefs '%s'␣
↪entry\n"
<6> arch/x86/kernel/hpet.c:144 _hpet_print_config "hpet: %s(%d):\n"
<6> init/do_mounts.c:605 prepare_namespace "Waiting for root device %s...\n"
<6> drivers/acpi/osl.c:1410 acpi_no_auto_serialize_setup "ACPI: auto-
↪serialization disabled\n"
```

, where the meaning is:

- **level**
  log level value: 0-7 for particular severity, -1 as default, 'c' as continuous line

without an explicit log level

- **flags**
  optional flags: currently only 'c' for KERN_CONT
- **filename:line**
  source filename and line number of the related printk() call. Note that there are many wrappers, for example, pr_warn(), pr_warn_once(), dev_warn().
- **function**
  function name where the printk() call is used.
- **format**
  format string

The extra information makes it a bit harder to find differences between various kernels. Especially the line number might change very often. On the other hand, it helps a lot to confirm that it is the same string or find the commit that is responsible for eventual changes.

1.6.2 printk() Is Not a Stable KABI

Several developers are afraid that exporting all these implementation details into the user space will transform particular printk() calls into KABI.

But it is exactly the opposite. printk() calls must _not_ be KABI. And the printk index helps user space tools to deal with this.

1.6.3 Subsystem specific printk wrappers

The printk index is generated using extra metadata that are stored in a dedicated .elf section ".printk_index". It is achieved using macro wrappers doing __printk_index_emit() together with the real printk() call. The same technique is used also for the metadata used by the dynamic debug feature.

The metadata are stored for a particular message only when it is printed using these special wrappers. It is implemented for the commonly used printk() calls, including, for example, pr_warn(), or pr_once().

Additional changes are necessary for various subsystem specific wrappers that call the original printk() via a common helper function. These needs their own wrappers adding __printk_index_emit().

Only few subsystem specific wrappers have been updated so far, for example, dev_printk(). As a result, the printk formats from some subsystes can be missing in the printk index.

1.6.4 Subsystem specific prefix

The macro pr_fmt() macro allows to define a prefix that is printed before the string generated by the related printk() calls.

Subsystem specific wrappers usually add even more complicated prefixes.

These prefixes can be stored into the printk index metadata by an optional parameter of __printk_index_emit(). The debugfs interface might then show the printk formats including these prefixes. For example, drivers/acpi/osl.c contains:

```
#define pr_fmt(fmt) "ACPI: OSL: " fmt

static int __init acpi_no_auto_serialize_setup(char *str)
{
      acpi_gbl_auto_serialize_methods = FALSE;
      pr_info("Auto-serialization disabled\n");

      return 1;
}
```

This results in the following printk index entry:

```
<6> drivers/acpi/osl.c:1410 acpi_no_auto_serialize_setup "ACPI: auto-
→serialization disabled\n"
```

It helps matching messages from the real log with printk index. Then the source file name, line number, and function name can be used to match the string with the source code.

1.7 Symbol Namespaces

The following document describes how to use Symbol Namespaces to structure the export surface of in-kernel symbols exported through the family of EXPORT_SYMBOL() macros.

1.7.1 1. Introduction

Symbol Namespaces have been introduced as a means to structure the export surface of the in-kernel API. It allows subsystem maintainers to partition their exported symbols into separate namespaces. That is useful for documentation purposes (think of the SUBSYSTEM_DEBUG namespace) as well as for limiting the availability of a set of symbols for use in other parts of the kernel. As of today, modules that make use of symbols exported into namespaces, are required to import the namespace. Otherwise the kernel will, depending on its configuration, reject loading the module or warn about a missing import.

1.7.2 2. How to define Symbol Namespaces

Symbols can be exported into namespace using different methods. All of them are changing the way EXPORT_SYMBOL and friends are instrumented to create ksymtab entries.

1.7.3 2.1 Using the EXPORT_SYMBOL macros

In addition to the macros EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL(), that allow exporting of kernel symbols to the kernel symbol table, variants of these are available to export symbols into a certain namespace: EXPORT_SYMBOL_NS() and EXPORT_SYMBOL_NS_GPL(). They take one additional argument: the namespace. Please note that due to macro expansion that argument needs to be a preprocessor symbol. E.g. to export the symbol `usb_stor_suspend` into the namespace `USB_STORAGE`, use:

```
EXPORT_SYMBOL_NS(usb_stor_suspend, USB_STORAGE);
```

The corresponding ksymtab entry struct `kernel_symbol` will have the member `namespace` set accordingly. A symbol that is exported without a namespace will refer to `NULL`. There is no default namespace if none is defined. `modpost` and kernel/module/main.c make use the namespace at build time or module load time, respectively.

1.7.4 2.2 Using the DEFAULT_SYMBOL_NAMESPACE define

Defining namespaces for all symbols of a subsystem can be very verbose and may become hard to maintain. Therefore a default define (DEFAULT_SYMBOL_NAMESPACE) is been provided, that, if set, will become the default for all EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL() macro expansions that do not specify a namespace.

There are multiple ways of specifying this define and it depends on the subsystem and the maintainer's preference, which one to use. The first option is to define the default namespace in the `Makefile` of the subsystem. E.g. to export all symbols defined in usb-common into the namespace USB_COMMON, add a line like this to drivers/usb/common/Makefile:

```
ccflags-y += -DDEFAULT_SYMBOL_NAMESPACE=USB_COMMON
```

That will affect all EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL() statements. A symbol exported with EXPORT_SYMBOL_NS() while this definition is present, will still be exported into the namespace that is passed as the namespace argument as this argument has preference over a default symbol namespace.

A second option to define the default namespace is directly in the compilation unit as preprocessor statement. The above example would then read:

```
#undef  DEFAULT_SYMBOL_NAMESPACE
#define DEFAULT_SYMBOL_NAMESPACE USB_COMMON
```

within the corresponding compilation unit before any EXPORT_SYMBOL macro is used.

1.7.5 3. How to use Symbols exported in Namespaces

In order to use symbols that are exported into namespaces, kernel modules need to explicitly import these namespaces. Otherwise the kernel might reject to load the module. The module code is required to use the macro MODULE_IMPORT_NS for the namespaces it uses symbols from. E.g. a module using the usb_stor_suspend symbol from above, needs to import the namespace USB_STORAGE using a statement like:

```
MODULE_IMPORT_NS(USB_STORAGE);
```

This will create a `modinfo` tag in the module for each imported namespace. This has the side effect, that the imported namespaces of a module can be inspected with modinfo:

```
$ modinfo drivers/usb/storage/ums-karma.ko
[...]
import_ns:      USB_STORAGE
[...]
```

It is advisable to add the MODULE_IMPORT_NS() statement close to other module metadata definitions like MODULE_AUTHOR() or MODULE_LICENSE(). Refer to section 5. for a way to create missing import statements automatically.

1.7.6 4. Loading Modules that use namespaced Symbols

At module loading time (e.g. `insmod`), the kernel will check each symbol referenced from the module for its availability and whether the namespace it might be exported to has been imported by the module. The default behaviour of the kernel is to reject loading modules that don't specify sufficient imports. An error will be logged and loading will be failed with EINVAL. In order to allow loading of modules that don't satisfy this precondition, a configuration option is available: Setting MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS=y will enable loading regardless, but will emit a warning.

1.7.7 5. Automatically creating MODULE_IMPORT_NS statements

Missing namespaces imports can easily be detected at build time. In fact, modpost will emit a warning if a module uses a symbol from a namespace without importing it. MODULE_IMPORT_NS() statements will usually be added at a definite location (along with other module meta data). To make the life of module authors (and subsystem maintainers) easier, a script and make target is available to fixup missing imports. Fixing missing imports can be done with:

```
$ make nsdeps
```

A typical scenario for module authors would be:

```
- write code that depends on a symbol from a not imported namespace
- ``make``
- notice the warning of modpost telling about a missing import
- run ``make nsdeps`` to add the import to the correct code location
```

For subsystem maintainers introducing a namespace, the steps are very similar. Again, `make nsdeps` will eventually add the missing namespace imports for in-tree modules:

```
- move or add symbols to a namespace (e.g. with EXPORT_SYMBOL_NS())
- ``make`` (preferably with an allmodconfig to cover all in-kernel
  modules)
- notice the warning of modpost telling about a missing import
- run ``make nsdeps`` to add the import to the correct code location
```

You can also run nsdeps for external module builds. A typical usage is:

```
$ make -C <path_to_kernel_src> M=$PWD nsdeps
```

1.8 Assembler Annotations

This document describes the new macros for annotation of data and code in assembly. In particular, it contains information about `SYM_FUNC_START`, `SYM_FUNC_END`, `SYM_CODE_START`, and similar.

1.8.1 Rationale

Some code like entries, trampolines, or boot code needs to be written in assembly. The same as in C, such code is grouped into functions and accompanied with data. Standard assemblers do not force users into precisely marking these pieces as code, data, or even specifying their length. Nevertheless, assemblers provide developers with such annotations to aid debuggers throughout assembly. On top of that, developers also want to mark some functions as *global* in order to be visible outside of their translation units.

Over time, the Linux kernel has adopted macros from various projects (like `binutils`) to facilitate such annotations. So for historic reasons, developers have been using `ENTRY`, `END`, `ENDPROC`, and other annotations in assembly. Due to the lack of their documentation, the macros are used in rather wrong contexts at some locations. Clearly, `ENTRY` was intended to denote the beginning of global symbols (be it data or code). `END` used to mark the end of data or end of special functions with *non-standard* calling convention. In contrast, `ENDPROC` should annotate only ends of *standard* functions.

When these macros are used correctly, they help assemblers generate a nice object with both sizes and types set correctly. For example, the result of `arch/x86/lib/putuser.S`:

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
 25: 0000000000000000    33 FUNC    GLOBAL DEFAULT    1 __put_user_1
 29: 0000000000000030    37 FUNC    GLOBAL DEFAULT    1 __put_user_2
 32: 0000000000000060    36 FUNC    GLOBAL DEFAULT    1 __put_user_4
 35: 0000000000000090    37 FUNC    GLOBAL DEFAULT    1 __put_user_8
```

This is not only important for debugging purposes. When there are properly annotated objects like this, tools can be run on them to generate more useful information. In particular, on properly annotated objects, `objtool` can be run to check and fix the object if needed. Currently, objtool can report missing frame pointer setup/destruction in functions. It can also automatically generate annotations for the ORC unwinder (Documentation/arch/x86/orc-unwinder.rst) for most code. Both of these are especially important to support reliable stack traces which are in turn necessary for kernel live patching (Documentation/livepatch/livepatch.rst).

1.8.2 Caveat and Discussion

As one might realize, there were only three macros previously. That is indeed insufficient to cover all the combinations of cases:

- standard/non-standard function
- code/data
- global/local symbol

There was a discussion and instead of extending the current ENTRY/END* macros, it was decided that brand new macros should be introduced instead:

```
So how about using macro names that actually show the purpose, instead
of importing all the crappy, historic, essentially randomly chosen
debug symbol macro names from the binutils and older kernels?
```

1.8.3 Macros Description

The new macros are prefixed with the SYM_ prefix and can be divided into three main groups:

1. SYM_FUNC_* -- to annotate C-like functions. This means functions with standard C calling conventions. For example, on x86, this means that the stack contains a return address at the predefined place and a return from the function can happen in a standard way. When frame pointers are enabled, save/restore of frame pointer shall happen at the start/end of a function, respectively, too.

   Checking tools like objtool should ensure such marked functions conform to these rules. The tools can also easily annotate these functions with debugging information (like *ORC data*) automatically.

2. SYM_CODE_* -- special functions called with special stack. Be it interrupt handlers with special stack content, trampolines, or startup functions.

   Checking tools mostly ignore checking of these functions. But some debug information still can be generated automatically. For correct debug data, this code needs hints like UNWIND_HINT_REGS provided by developers.

3. SYM_DATA* -- obviously data belonging to .data sections and not to .text. Data do not contain instructions, so they have to be treated specially by the tools: they should not treat the bytes as instructions, nor assign any debug information to them.

Instruction Macros

This section covers SYM_FUNC_* and SYM_CODE_* enumerated above.

objtool requires that all code must be contained in an ELF symbol. Symbol names that have a .L prefix do not emit symbol table entries. .L prefixed symbols can be used within a code region, but should be avoided for denoting a range of code via SYM_*_START/END annotations.

- SYM_FUNC_START and SYM_FUNC_START_LOCAL are supposed to be **the most frequent markings**. They are used for functions with standard calling conventions -- global and local. Like in C, they both align the functions to architecture specific __ALIGN bytes. There are also _NOALIGN variants for special cases where developers do not want this implicit alignment.

  SYM_FUNC_START_WEAK and SYM_FUNC_START_WEAK_NOALIGN markings are also offered as an assembler counterpart to the *weak* attribute known from C.

  All of these **shall** be coupled with SYM_FUNC_END. First, it marks the sequence of instructions as a function and computes its size to the generated object file. Second, it also eases checking and processing such object files as the tools can trivially find exact function boundaries.

  So in most cases, developers should write something like in the following example, having some asm instructions in between the macros, of course:

  ```
  SYM_FUNC_START(memset)
      ... asm insns ...
  SYM_FUNC_END(memset)
  ```

  In fact, this kind of annotation corresponds to the now deprecated ENTRY and ENDPROC macros.

- SYM_FUNC_ALIAS, SYM_FUNC_ALIAS_LOCAL, and SYM_FUNC_ALIAS_WEAK can be used to define multiple names for a function. The typical use is:

  ```
  SYM_FUNC_START(__memset)
      ... asm insns ...
  SYN_FUNC_END(__memset)
  SYM_FUNC_ALIAS(memset, __memset)
  ```

  In this example, one can call __memset or memset with the same result, except the debug information for the instructions is generated to the object file only once -- for the non-ALIAS case.

- SYM_CODE_START and SYM_CODE_START_LOCAL should be used only in special cases -- if you know what you are doing. This is used exclusively for interrupt handlers and similar where the calling convention is not the C one. _NOALIGN variants exist too. The use is the same as for the FUNC category above:

  ```
  SYM_CODE_START_LOCAL(bad_put_user)
      ... asm insns ...
  SYM_CODE_END(bad_put_user)
  ```

  Again, every SYM_CODE_START* **shall** be coupled by SYM_CODE_END.

To some extent, this category corresponds to deprecated ENTRY and END. Except END had several other meanings too.

- SYM_INNER_LABEL* is used to denote a label inside some SYM_{CODE,FUNC}_START and SYM_{CODE,FUNC}_END. They are very similar to C labels, except they can be made global. An example of use:

```
SYM_CODE_START(ftrace_caller)
    /* save_mcount_regs fills in first two parameters */
    ...

SYM_INNER_LABEL(ftrace_caller_op_ptr, SYM_L_GLOBAL)
    /* Load the ftrace_ops into the 3rd parameter */
    ...

SYM_INNER_LABEL(ftrace_call, SYM_L_GLOBAL)
    call ftrace_stub
    ...
    retq
SYM_CODE_END(ftrace_caller)
```

Data Macros

Similar to instructions, there is a couple of macros to describe data in the assembly.

- SYM_DATA_START and SYM_DATA_START_LOCAL mark the start of some data and shall be used in conjunction with either SYM_DATA_END, or SYM_DATA_END_LABEL. The latter adds also a label to the end, so that people can use lstack and (local) lstack_end in the following example:

```
SYM_DATA_START_LOCAL(lstack)
    .skip 4096
SYM_DATA_END_LABEL(lstack, SYM_L_LOCAL, lstack_end)
```

- SYM_DATA and SYM_DATA_LOCAL are variants for simple, mostly one-line data:

```
SYM_DATA(HEAP,     .long rm_heap)
SYM_DATA(heap_end, .long rm_stack)
```

  In the end, they expand to SYM_DATA_START with SYM_DATA_END internally.

Support Macros

All the above reduce themselves to some invocation of SYM_START, SYM_END, or SYM_ENTRY at last. Normally, developers should avoid using these.

Further, in the above examples, one could see SYM_L_LOCAL. There are also SYM_L_GLOBAL and SYM_L_WEAK. All are intended to denote linkage of a symbol marked by them. They are used either in _LABEL variants of the earlier macros, or in SYM_START.

Overriding Macros

Architecture can also override any of the macros in their own `asm/linkage.h`, including macros specifying the type of a symbol (`SYM_T_FUNC`, `SYM_T_OBJECT`, and `SYM_T_NONE`). As every macro described in this file is surrounded by `#ifdef` + `#endif`, it is enough to define the macros differently in the aforementioned architecture-dependent header.

CHAPTER

TWO

DATA STRUCTURES AND LOW-LEVEL UTILITIES

Library functionality that is used throughout the kernel.

2.1 Everything you never wanted to know about kobjects, ksets, and ktypes

Author
Greg Kroah-Hartman <gregkh@linuxfoundation.org>

Last updated
December 19, 2007

Based on an original article by Jon Corbet for lwn.net written October 1, 2003 and located at https://lwn.net/Articles/51437/

Part of the difficulty in understanding the driver model - and the kobject abstraction upon which it is built - is that there is no obvious starting place. Dealing with kobjects requires understanding a few different types, all of which make reference to each other. In an attempt to make things easier, we'll take a multi-pass approach, starting with vague terms and adding detail as we go. To that end, here are some quick definitions of some terms we will be working with.

- A kobject is an object of type struct kobject. Kobjects have a name and a reference count. A kobject also has a parent pointer (allowing objects to be arranged into hierarchies), a specific type, and, usually, a representation in the sysfs virtual filesystem.

  Kobjects are generally not interesting on their own; instead, they are usually embedded within some other structure which contains the stuff the code is really interested in.

  No structure should **EVER** have more than one kobject embedded within it. If it does, the reference counting for the object is sure to be messed up and incorrect, and your code will be buggy. So do not do this.

- A ktype is the type of object that embeds a kobject. Every structure that embeds a kobject needs a corresponding ktype. The ktype controls what happens to the kobject when it is created and destroyed.

- A kset is a group of kobjects. These kobjects can be of the same ktype or belong to different ktypes. The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, but you can safely ignore that implementation detail as the kset core code handles this kobject automatically.

  When you see a sysfs directory full of other directories, generally each of those directories corresponds to a kobject in the same kset.

We'll look at how to create and manipulate all of these types. A bottom-up approach will be taken, so we'll go back to kobjects.

2.1.1 Embedding kobjects

It is rare for kernel code to create a standalone kobject, with one major exception explained below. Instead, kobjects are used to control access to a larger, domain-specific object. To this end, kobjects will be found embedded in other structures. If you are used to thinking of things in object-oriented terms, kobjects can be seen as a top-level, abstract class from which other classes are derived. A kobject implements a set of capabilities which are not particularly useful by themselves, but are nice to have in other objects. The C language does not allow for the direct expression of inheritance, so other techniques - such as structure embedding - must be used.

(As an aside, for those familiar with the kernel linked list implementation, this is analogous as to how "list_head" structs are rarely useful on their own, but are invariably found embedded in the larger objects of interest.)

So, for example, the UIO code in `drivers/uio/uio.c` has a structure that defines the memory region associated with a uio device:

```
struct uio_map {
        struct kobject kobj;
        struct uio_mem *mem;
};
```

If you have a struct uio_map structure, finding its embedded kobject is just a matter of using the kobj member. Code that works with kobjects will often have the opposite problem, however: given a struct kobject pointer, what is the pointer to the containing structure? You must avoid tricks (such as assuming that the kobject is at the beginning of the structure) and, instead, use the container_of() macro, found in `<linux/kernel.h>`:

```
container_of(ptr, type, member)
```

where:

- `ptr` is the pointer to the embedded kobject,
- `type` is the type of the containing structure, and
- `member` is the name of the structure field to which `pointer` points.

The return value from container_of() is a pointer to the corresponding container type. So, for example, a pointer kp to a struct kobject embedded **within** a struct uio_map could be converted to a pointer to the **containing** uio_map structure with:

```
struct uio_map *u_map = container_of(kp, struct uio_map, kobj);
```

For convenience, programmers often define a simple macro for **back-casting** kobject pointers to the containing type. Exactly this happens in the earlier `drivers/uio/uio.c`, as you can see here:

```
struct uio_map {
        struct kobject kobj;
```

```
        struct uio_mem *mem;
};

#define to_map(map) container_of(map, struct uio_map, kobj)
```

where the macro argument "map" is a pointer to the struct kobject in question. That macro is subsequently invoked with:

```
struct uio_map *map = to_map(kobj);
```

2.1.2 Initialization of kobjects

Code which creates a kobject must, of course, initialize that object. Some of the internal fields are setup with a (mandatory) call to kobject_init():

```
void kobject_init(struct kobject *kobj, const struct kobj_type *ktype);
```

The ktype is required for a kobject to be created properly, as every kobject must have an associated kobj_type. After calling kobject_init(), to register the kobject with sysfs, the function kobject_add() must be called:

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
                const char *fmt, ...);
```

This sets up the parent of the kobject and the name for the kobject properly. If the kobject is to be associated with a specific kset, kobj->kset must be assigned before calling kobject_add(). If a kset is associated with a kobject, then the parent for the kobject can be set to NULL in the call to kobject_add() and then the kobject's parent will be the kset itself.

As the name of the kobject is set when it is added to the kernel, the name of the kobject should never be manipulated directly. If you must change the name of the kobject, call kobject_rename():

```
int kobject_rename(struct kobject *kobj, const char *new_name);
```

kobject_rename() does not perform any locking or have a solid notion of what names are valid so the caller must provide their own sanity checking and serialization.

There is a function called kobject_set_name() but that is legacy cruft and is being removed. If your code needs to call this function, it is incorrect and needs to be fixed.

To properly access the name of the kobject, use the function kobject_name():

```
const char *kobject_name(const struct kobject * kobj);
```

There is a helper function to both initialize and add the kobject to the kernel at the same time, called surprisingly enough kobject_init_and_add():

```
int kobject_init_and_add(struct kobject *kobj, const struct kobj_type *ktype,
                         struct kobject *parent, const char *fmt, ...);
```

The arguments are the same as the individual kobject_init() and kobject_add() functions described above.

2.1.3 Uevents

After a kobject has been registered with the kobject core, you need to announce to the world that it has been created. This can be done with a call to kobject_uevent():

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action);
```

Use the **KOBJ_ADD** action for when the kobject is first added to the kernel. This should be done only after any attributes or children of the kobject have been initialized properly, as userspace will instantly start to look for them when this call happens.

When the kobject is removed from the kernel (details on how to do that are below), the uevent for **KOBJ_REMOVE** will be automatically created by the kobject core, so the caller does not have to worry about doing that by hand.

2.1.4 Reference counts

One of the key functions of a kobject is to serve as a reference counter for the object in which it is embedded. As long as references to the object exist, the object (and the code which supports it) must continue to exist. The low-level functions for manipulating a kobject's reference counts are:

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

A successful call to kobject_get() will increment the kobject's reference counter and return the pointer to the kobject.

When a reference is released, the call to kobject_put() will decrement the reference count and, possibly, free the object. Note that kobject_init() sets the reference count to one, so the code which sets up the kobject will need to do a kobject_put() eventually to release that reference.

Because kobjects are dynamic, they must not be declared statically or on the stack, but instead, always allocated dynamically. Future versions of the kernel will contain a run-time check for kobjects that are created statically and will warn the developer of this improper usage.

If all that you want to use a kobject for is to provide a reference counter for your structure, please use the struct kref instead; a kobject would be overkill. For more information on how to use struct kref, please see the file *Adding reference counters (krefs) to kernel objects* in the Linux kernel source tree.

2.1.5 Creating "simple" kobjects

Sometimes all that a developer wants is a way to create a simple directory in the sysfs hierarchy, and not have to mess with the whole complication of ksets, show and store functions, and other details. This is the one exception where a single kobject should be created. To create such an entry, use the function:

```
struct kobject *kobject_create_and_add(const char *name, struct kobject␣
↪*parent);
```

This function will create a kobject and place it in sysfs in the location underneath the specified parent kobject. To create simple attributes associated with this kobject, use:

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

or:

```
int sysfs_create_group(struct kobject *kobj, const struct attribute_group␣
→*grp);
```

Both types of attributes used here, with a kobject that has been created with the kobject_create_and_add(), can be of type kobj_attribute, so no special custom attribute is needed to be created.

See the example module, `samples/kobject/kobject-example.c` for an implementation of a simple kobject and attributes.

2.1.6 ktypes and release methods

One important thing still missing from the discussion is what happens to a kobject when its reference count reaches zero. The code which created the kobject generally does not know when that will happen; if it did, there would be little point in using a kobject in the first place. Even predictable object lifecycles become more complicated when sysfs is brought in as other portions of the kernel can get a reference on any kobject that is registered in the system.

The end result is that a structure protected by a kobject cannot be freed before its reference count goes to zero. The reference count is not under the direct control of the code which created the kobject. So that code must be notified asynchronously whenever the last reference to one of its kobjects goes away.

Once you registered your kobject via kobject_add(), you must never use kfree() to free it directly. The only safe way is to use kobject_put(). It is good practice to always use kobject_put() after kobject_init() to avoid errors creeping in.

This notification is done through a kobject's release() method. Usually such a method has a form like:

```
void my_object_release(struct kobject *kobj)
{
        struct my_object *mine = container_of(kobj, struct my_object, kobj);

        /* Perform any additional cleanup on this object, then... */
        kfree(mine);
}
```

One important point cannot be overstated: every kobject must have a release() method, and the kobject must persist (in a consistent state) until that method is called. If these constraints are not met, the code is flawed. Note that the kernel will warn you if you forget to provide a release() method. Do not try to get rid of this warning by providing an "empty" release function.

If all your cleanup function needs to do is call kfree(), then you must create a wrapper function which uses container_of() to upcast to the correct type (as shown in the example above) and then calls kfree() on the overall structure.

Note, the name of the kobject is available in the release function, but it must NOT be changed within this callback. Otherwise there will be a memory leak in the kobject core, which makes people unhappy.

Interestingly, the release() method is not stored in the kobject itself; instead, it is associated with the ktype. So let us introduce struct kobj_type:

```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        const struct attribute_group **default_groups;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject␣
↪*kobj);
        const void *(*namespace)(struct kobject *kobj);
        void (*get_ownership)(struct kobject *kobj, kuid_t *uid, kgid_t *gid);
};
```

This structure is used to describe a particular type of kobject (or, more correctly, of containing object). Every kobject needs to have an associated kobj_type structure; a pointer to that structure must be specified when you call kobject_init() or kobject_init_and_add().

The release field in struct kobj_type is, of course, a pointer to the release() method for this type of kobject. The other two fields (sysfs_ops and default_groups) control how objects of this type are represented in sysfs; they are beyond the scope of this document.

The default_groups pointer is a list of default attributes that will be automatically created for any kobject that is registered with this ktype.

2.1.7 ksets

A kset is merely a collection of kobjects that want to be associated with each other. There is no restriction that they be of the same ktype, but be very careful if they are not.

A kset serves these functions:

- It serves as a bag containing a group of objects. A kset can be used by the kernel to track "all block devices" or "all PCI device drivers."
- A kset is also a subdirectory in sysfs, where the associated kobjects with the kset can show up. Every kset contains a kobject which can be set up to be the parent of other kobjects; the top-level directories of the sysfs hierarchy are constructed in this way.
- Ksets can support the "hotplugging" of kobjects and influence how uevent events are reported to user space.

In object-oriented terms, "kset" is the top-level container class; ksets contain their own kobject, but that kobject is managed by the kset code and should not be manipulated by any other user.

A kset keeps its children in a standard kernel linked list. Kobjects point back to their containing kset via their kset field. In almost all cases, the kobjects belonging to a kset have that kset (or, strictly, its embedded kobject) in their parent.

As a kset contains a kobject within it, it should always be dynamically created and never declared statically or on the stack. To create a new kset use:

```
struct kset *kset_create_and_add(const char *name,
                                 const struct kset_uevent_ops *uevent_ops,
                                 struct kobject *parent_kobj);
```

When you are finished with the kset, call:

```
void kset_unregister(struct kset *k);
```

to destroy it. This removes the kset from sysfs and decrements its reference count. When the reference count goes to zero, the kset will be released. Because other references to the kset may still exist, the release may happen after kset_unregister() returns.

An example of using a kset can be seen in the `samples/kobject/kset-example.c` file in the kernel tree.

If a kset wishes to control the uevent operations of the kobjects associated with it, it can use the struct kset_uevent_ops to handle it:

```
struct kset_uevent_ops {
        int (* const filter)(struct kobject *kobj);
        const char *(* const name)(struct kobject *kobj);
        int (* const uevent)(struct kobject *kobj, struct kobj_uevent_env␣
↪*env);
};
```

The filter function allows a kset to prevent a uevent from being emitted to userspace for a specific kobject. If the function returns 0, the uevent will not be emitted.

The name function will be called to override the default name of the kset that the uevent sends to userspace. By default, the name will be the same as the kset itself, but this function, if present, can override that name.

The uevent function will be called when the uevent is about to be sent to userspace to allow more environment variables to be added to the uevent.

One might ask how, exactly, a kobject is added to a kset, given that no functions which perform that function have been presented. The answer is that this task is handled by kobject_add(). When a kobject is passed to kobject_add(), its kset member should point to the kset to which the kobject will belong. kobject_add() will handle the rest.

If the kobject belonging to a kset has no parent kobject set, it will be added to the kset's directory. Not all members of a kset do necessarily live in the kset directory. If an explicit parent kobject is assigned before the kobject is added, the kobject is registered with the kset, but added below the parent kobject.

2.1.8 Kobject removal

After a kobject has been registered with the kobject core successfully, it must be cleaned up when the code is finished with it. To do that, call kobject_put(). By doing this, the kobject core will automatically clean up all of the memory allocated by this kobject. If a `KOBJ_ADD` uevent has been sent for the object, a corresponding `KOBJ_REMOVE` uevent will be sent, and any other sysfs housekeeping will be handled for the caller properly.

If you need to do a two-stage delete of the kobject (say you are not allowed to sleep when you need to destroy the object), then call kobject_del() which will unregister the kobject from sysfs. This makes the kobject "invisible", but it is not cleaned up, and the reference count of the object is still the same. At a later time call kobject_put() to finish the cleanup of the memory associated with the kobject.

kobject_del() can be used to drop the reference to the parent object, if circular references are constructed. It is valid in some cases, that a parent objects references a child. Circular references _must_ be broken with an explicit call to kobject_del(), so that a release functions will be called, and the objects in the former circle release each other.

2.1.9 Example code to copy from

For a more complete example of using ksets and kobjects properly, see the example programs `samples/kobject/{kobject-example.c,kset-example.c}`, which will be built as loadable modules if you select `CONFIG_SAMPLE_KOBJECT`.

2.2 Adding reference counters (krefs) to kernel objects

Author
Corey Minyard <minyard@acm.org>

Author
Thomas Hellstrom <thellstrom@vmware.com>

A lot of this was lifted from Greg Kroah-Hartman's 2004 OLS paper and presentation on krefs, which can be found at:

- http://www.kroah.com/linux/talks/ols_2004_kref_paper/Reprint-Kroah-Hartman-OLS2004.pdf
- http://www.kroah.com/linux/talks/ols_2004_kref_talk/

2.2.1 Introduction

krefs allow you to add reference counters to your objects. If you have objects that are used in multiple places and passed around, and you don't have refcounts, your code is almost certainly broken. If you want refcounts, krefs are the way to go.

To use a kref, add one to your data structures like:

```
struct my_data
{
    .
    .
    struct kref refcount;
    .
    .
};
```

The kref can occur anywhere within the data structure.

2.2.2 Initialization

You must initialize the kref after you allocate it. To do this, call kref_init as so:

```
struct my_data *data;

data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data)
       return -ENOMEM;
kref_init(&data->refcount);
```

This sets the refcount in the kref to 1.

2.2.3 Kref rules

Once you have an initialized kref, you must follow the following rules:

1) If you make a non-temporary copy of a pointer, especially if it can be passed to another thread of execution, you must increment the refcount with kref_get() before passing it off:

   ```
   kref_get(&data->refcount);
   ```

   If you already have a valid pointer to a kref-ed structure (the refcount cannot go to zero) you may do this without a lock.

2) When you are done with a pointer, you must call kref_put():

   ```
   kref_put(&data->refcount, data_release);
   ```

   If this is the last reference to the pointer, the release routine will be called. If the code never tries to get a valid pointer to a kref-ed structure without already holding a valid pointer, it is safe to do this without a lock.

3) If the code attempts to gain a reference to a kref-ed structure without already holding a valid pointer, it must serialize access where a kref_put() cannot occur during the kref_get(), and the structure must remain valid during the kref_get().

For example, if you allocate some data and then pass it to another thread to process:

```
void data_release(struct kref *ref)
{
    struct my_data *data = container_of(ref, struct my_data, refcount);
    kfree(data);
}

void more_data_handling(void *cb_data)
{
    struct my_data *data = cb_data;
    .
    . do stuff with data here
    .
    kref_put(&data->refcount, data_release);
}
```

```
int my_data_handler(void)
{
    int rv = 0;
    struct my_data *data;
    struct task_struct *task;
    data = kmalloc(sizeof(*data), GFP_KERNEL);
    if (!data)
            return -ENOMEM;
    kref_init(&data->refcount);

    kref_get(&data->refcount);
    task = kthread_run(more_data_handling, data, "more_data_handling");
    if (task == ERR_PTR(-ENOMEM)) {
            rv = -ENOMEM;
            kref_put(&data->refcount, data_release);
            goto out;
    }

    .
    . do stuff with data here
    .
out:
    kref_put(&data->refcount, data_release);
    return rv;
}
```

This way, it doesn't matter what order the two threads handle the data, the kref_put() handles knowing when the data is not referenced any more and releasing it. The kref_get() does not require a lock, since we already have a valid pointer that we own a refcount for. The put needs no lock because nothing tries to get the data without already holding a pointer.

In the above example, kref_put() will be called 2 times in both success and error paths. This is necessary because the reference count got incremented 2 times by kref_init() and kref_get().

Note that the "before" in rule 1 is very important. You should never do something like:

```
task = kthread_run(more_data_handling, data, "more_data_handling");
if (task == ERR_PTR(-ENOMEM)) {
        rv = -ENOMEM;
        goto out;
} else
        /* BAD BAD BAD - get is after the handoff */
        kref_get(&data->refcount);
```

Don't assume you know what you are doing and use the above construct. First of all, you may not know what you are doing. Second, you may know what you are doing (there are some situations where locking is involved where the above may be legal) but someone else who doesn't know what they are doing may change the code or copy the code. It's bad style. Don't do it.

There are some situations where you can optimize the gets and puts. For instance, if you are done with an object and enqueuing it for something else or passing it off to something else, there is no reason to do a get then a put:

```
/* Silly extra get and put */
kref_get(&obj->ref);
enqueue(obj);
kref_put(&obj->ref, obj_cleanup);
```

Just do the enqueue. A comment about this is always welcome:

```
enqueue(obj);
/* We are done with obj, so we pass our refcount off
   to the queue.  DON'T TOUCH obj AFTER HERE! */
```

The last rule (rule 3) is the nastiest one to handle. Say, for instance, you have a list of items that are each kref-ed, and you wish to get the first one. You can't just pull the first item off the list and kref_get() it. That violates rule 3 because you are not already holding a valid pointer. You must add a mutex (or some other lock). For instance:

```
static DEFINE_MUTEX(mutex);
static LIST_HEAD(q);
struct my_data
{
        struct kref      refcount;
        struct list_head link;
};

static struct my_data *get_entry()
{
        struct my_data *entry = NULL;
        mutex_lock(&mutex);
        if (!list_empty(&q)) {
                entry = container_of(q.next, struct my_data, link);
                kref_get(&entry->refcount);
        }
        mutex_unlock(&mutex);
        return entry;
}

static void release_entry(struct kref *ref)
{
        struct my_data *entry = container_of(ref, struct my_data, refcount);

        list_del(&entry->link);
        kfree(entry);
}

static void put_entry(struct my_data *entry)
{
        mutex_lock(&mutex);
        kref_put(&entry->refcount, release_entry);
        mutex_unlock(&mutex);
}
```

The kref_put() return value is useful if you do not want to hold the lock during the whole release operation. Say you didn't want to call kfree() with the lock held in the example above (since it is kind of pointless to do so). You could use kref_put() as follows:

```
static void release_entry(struct kref *ref)
{
        /* All work is done after the return from kref_put(). */
}

static void put_entry(struct my_data *entry)
{
        mutex_lock(&mutex);
        if (kref_put(&entry->refcount, release_entry)) {
                list_del(&entry->link);
                mutex_unlock(&mutex);
                kfree(entry);
        } else
                mutex_unlock(&mutex);
}
```

This is really more useful if you have to call other routines as part of the free operations that could take a long time or might claim the same lock. Note that doing everything in the release routine is still preferred as it is a little neater.

The above example could also be optimized using kref_get_unless_zero() in the following way:

```
static struct my_data *get_entry()
{
        struct my_data *entry = NULL;
        mutex_lock(&mutex);
        if (!list_empty(&q)) {
                entry = container_of(q.next, struct my_data, link);
                if (!kref_get_unless_zero(&entry->refcount))
                        entry = NULL;
        }
        mutex_unlock(&mutex);
        return entry;
}

static void release_entry(struct kref *ref)
{
        struct my_data *entry = container_of(ref, struct my_data, refcount);

        mutex_lock(&mutex);
        list_del(&entry->link);
        mutex_unlock(&mutex);
        kfree(entry);
}

static void put_entry(struct my_data *entry)
{
        kref_put(&entry->refcount, release_entry);
```

```
}
```

Which is useful to remove the mutex lock around kref_put() in put_entry(), but it's important that kref_get_unless_zero is enclosed in the same critical section that finds the entry in the lookup table, otherwise kref_get_unless_zero may reference already freed memory. Note that it is illegal to use kref_get_unless_zero without checking its return value. If you are sure (by already having a valid pointer) that kref_get_unless_zero() will return true, then use kref_get() instead.

2.2.4 Krefs and RCU

The function kref_get_unless_zero also makes it possible to use rcu locking for lookups in the above example:

```
struct my_data
{
        struct rcu_head rhead;
        .
        struct kref refcount;
        .
        .
};

static struct my_data *get_entry_rcu()
{
        struct my_data *entry = NULL;
        rcu_read_lock();
        if (!list_empty(&q)) {
                entry = container_of(q.next, struct my_data, link);
                if (!kref_get_unless_zero(&entry->refcount))
                        entry = NULL;
        }
        rcu_read_unlock();
        return entry;
}

static void release_entry_rcu(struct kref *ref)
{
        struct my_data *entry = container_of(ref, struct my_data, refcount);

        mutex_lock(&mutex);
        list_del_rcu(&entry->link);
        mutex_unlock(&mutex);
        kfree_rcu(entry, rhead);
}

static void put_entry(struct my_data *entry)
{
        kref_put(&entry->refcount, release_entry_rcu);
}
```

But note that the struct kref member needs to remain in valid memory for a rcu grace period after release_entry_rcu was called. That can be accomplished by using kfree_rcu(entry, rhead) as done above, or by calling synchronize_rcu() before using kfree, but note that synchronize_rcu() may sleep for a substantial amount of time.

2.3 Generic Associative Array Implementation

2.3.1 Overview

This associative array implementation is an object container with the following properties:

1. Objects are opaque pointers. The implementation does not care where they point (if anywhere) or what they point to (if anything).

   Note: Pointers to objects _must_ be zero in the least significant bit.

2. Objects do not need to contain linkage blocks for use by the array. This permits an object to be located in multiple arrays simultaneously. Rather, the array is made up of metadata blocks that point to objects.
3. Objects require index keys to locate them within the array.
4. Index keys must be unique. Inserting an object with the same key as one already in the array will replace the old object.
5. Index keys can be of any length and can be of different lengths.
6. Index keys should encode the length early on, before any variation due to length is seen.
7. Index keys can include a hash to scatter objects throughout the array.
8. The array can iterated over. The objects will not necessarily come out in key order.
9. The array can be iterated over while it is being modified, provided the RCU readlock is being held by the iterator. Note, however, under these circumstances, some objects may be seen more than once. If this is a problem, the iterator should lock against modification. Objects will not be missed, however, unless deleted.
10. Objects in the array can be looked up by means of their index key.
11. Objects can be looked up while the array is being modified, provided the RCU readlock is being held by the thread doing the look up.

The implementation uses a tree of 16-pointer nodes internally that are indexed on each level by nibbles from the index key in the same manner as in a radix tree. To improve memory efficiency, shortcuts can be emplaced to skip over what would otherwise be a series of single-occupancy nodes. Further, nodes pack leaf object pointers into spare space in the node rather than making an extra branch until as such time an object needs to be added to a full node.

2.3.2 The Public API

The public API can be found in `<linux/assoc_array.h>`. The associative array is rooted on the following structure:

```
struct assoc_array {
        ...
};
```

The code is selected by enabling `CONFIG_ASSOCIATIVE_ARRAY` with:

```
./script/config -e ASSOCIATIVE_ARRAY
```

Edit Script

The insertion and deletion functions produce an 'edit script' that can later be applied to effect the changes without risking `ENOMEM`. This retains the preallocated metadata blocks that will be installed in the internal tree and keeps track of the metadata blocks that will be removed from the tree when the script is applied.

This is also used to keep track of dead blocks and dead objects after the script has been applied so that they can be freed later. The freeing is done after an RCU grace period has passed - thus allowing access functions to proceed under the RCU read lock.

The script appears as outside of the API as a pointer of the type:

```
struct assoc_array_edit;
```

There are two functions for dealing with the script:

1. Apply an edit script:

   ```
   void assoc_array_apply_edit(struct assoc_array_edit *edit);
   ```

This will perform the edit functions, interpolating various write barriers to permit accesses under the RCU read lock to continue. The edit script will then be passed to `call_rcu()` to free it and any dead stuff it points to.

2. Cancel an edit script:

   ```
   void assoc_array_cancel_edit(struct assoc_array_edit *edit);
   ```

This frees the edit script and all preallocated memory immediately. If this was for insertion, the new object is _not_ released by this function, but must rather be released by the caller.

These functions are guaranteed not to fail.

Operations Table

Various functions take a table of operations:

```
struct assoc_array_ops {
        ...
};
```

This points to a number of methods, all of which need to be provided:

1. Get a chunk of index key from caller data:

   ```
   unsigned long (*get_key_chunk)(const void *index_key, int level);
   ```

This should return a chunk of caller-supplied index key starting at the *bit* position given by the level argument. The level argument will be a multiple of `ASSOC_ARRAY_KEY_CHUNK_SIZE` and the function should return `ASSOC_ARRAY_KEY_CHUNK_SIZE bits`. No error is possible.

2. Get a chunk of an object's index key:

   ```
   unsigned long (*get_object_key_chunk)(const void *object, int level);
   ```

As the previous function, but gets its data from an object in the array rather than from a caller-supplied index key.

3. See if this is the object we're looking for:

   ```
   bool (*compare_object)(const void *object, const void *index_key);
   ```

Compare the object against an index key and return `true` if it matches and `false` if it doesn't.

4. Diff the index keys of two objects:

   ```
   int (*diff_objects)(const void *object, const void *index_key);
   ```

Return the bit position at which the index key of the specified object differs from the given index key or -1 if they are the same.

5. Free an object:

   ```
   void (*free_object)(void *object);
   ```

Free the specified object. Note that this may be called an RCU grace period after `assoc_array_apply_edit()` was called, so `synchronize_rcu()` may be necessary on module unloading.

Manipulation Functions

There are a number of functions for manipulating an associative array:

1. Initialise an associative array:

```
void assoc_array_init(struct assoc_array *array);
```

This initialises the base structure for an associative array. It can't fail.

2. Insert/replace an object in an associative array:

```
struct assoc_array_edit *
assoc_array_insert(struct assoc_array *array,
                   const struct assoc_array_ops *ops,
                   const void *index_key,
                   void *object);
```

This inserts the given object into the array. Note that the least significant bit of the pointer must be zero as it's used to type-mark pointers internally.

If an object already exists for that key then it will be replaced with the new object and the old one will be freed automatically.

The `index_key` argument should hold index key information and is passed to the methods in the ops table when they are called.

This function makes no alteration to the array itself, but rather returns an edit script that must be applied. `-ENOMEM` is returned in the case of an out-of-memory error.

The caller should lock exclusively against other modifiers of the array.

3. Delete an object from an associative array:

```
struct assoc_array_edit *
assoc_array_delete(struct assoc_array *array,
                   const struct assoc_array_ops *ops,
                   const void *index_key);
```

This deletes an object that matches the specified data from the array.

The `index_key` argument should hold index key information and is passed to the methods in the ops table when they are called.

This function makes no alteration to the array itself, but rather returns an edit script that must be applied. `-ENOMEM` is returned in the case of an out-of-memory error. `NULL` will be returned if the specified object is not found within the array.

The caller should lock exclusively against other modifiers of the array.

4. Delete all objects from an associative array:

```
struct assoc_array_edit *
assoc_array_clear(struct assoc_array *array,
                  const struct assoc_array_ops *ops);
```

This deletes all the objects from an associative array and leaves it completely empty.

This function makes no alteration to the array itself, but rather returns an edit script that must be applied. -ENOMEM is returned in the case of an out-of-memory error.

The caller should lock exclusively against other modifiers of the array.

5. Destroy an associative array, deleting all objects:

```
void assoc_array_destroy(struct assoc_array *array,
                         const struct assoc_array_ops *ops);
```

This destroys the contents of the associative array and leaves it completely empty. It is not permitted for another thread to be traversing the array under the RCU read lock at the same time as this function is destroying it as no RCU deferral is performed on memory release - something that would require memory to be allocated.

The caller should lock exclusively against other modifiers and accessors of the array.

6. Garbage collect an associative array:

```
int assoc_array_gc(struct assoc_array *array,
                   const struct assoc_array_ops *ops,
                   bool (*iterator)(void *object, void *iterator_data),
                   void *iterator_data);
```

This iterates over the objects in an associative array and passes each one to `iterator()`. If `iterator()` returns `true`, the object is kept. If it returns `false`, the object will be freed. If the `iterator()` function returns `true`, it must perform any appropriate refcount incrementing on the object before returning.

The internal tree will be packed down if possible as part of the iteration to reduce the number of nodes in it.

The `iterator_data` is passed directly to `iterator()` and is otherwise ignored by the function.

The function will return `0` if successful and `-ENOMEM` if there wasn't enough memory.

It is possible for other threads to iterate over or search the array under the RCU read lock while this function is in progress. The caller should lock exclusively against other modifiers of the array.

Access Functions

There are two functions for accessing an associative array:

1. Iterate over all the objects in an associative array:

```
int assoc_array_iterate(const struct assoc_array *array,
                        int (*iterator)(const void *object,
                                        void *iterator_data),
                        void *iterator_data);
```

This passes each object in the array to the iterator callback function. `iterator_data` is private data for that function.

This may be used on an array at the same time as the array is being modified, provided the RCU read lock is held. Under such circumstances, it is possible for the iteration function to see some objects twice. If this is a problem, then modification should be locked against. The iteration algorithm should not, however, miss any objects.

The function will return `0` if no objects were in the array or else it will return the result of the last iterator function called. Iteration stops immediately if any call to the iteration function results in a non-zero return.

2. Find an object in an associative array:

```
void *assoc_array_find(const struct assoc_array *array,
                       const struct assoc_array_ops *ops,
                       const void *index_key);
```

This walks through the array's internal tree directly to the object specified by the index key..

This may be used on an array at the same time as the array is being modified, provided the RCU read lock is held.

The function will return the object if found (and set `*_type` to the object type) or will return `NULL` if the object was not found.

Index Key Form

The index key can be of any form, but since the algorithms aren't told how long the key is, it is strongly recommended that the index key includes its length very early on before any variation due to the length would have an effect on comparisons.

This will cause leaves with different length keys to scatter away from each other - and those with the same length keys to cluster together.

It is also recommended that the index key begin with a hash of the rest of the key to maximise scattering throughout keyspace.

The better the scattering, the wider and lower the internal tree will be.

Poor scattering isn't too much of a problem as there are shortcuts and nodes can contain mixtures of leaves and metadata pointers.

The index key is read in chunks of machine word. Each chunk is subdivided into one nibble (4 bits) per level, so on a 32-bit CPU this is good for 8 levels and on a 64-bit CPU, 16 levels. Unless the scattering is really poor, it is unlikely that more than one word of any particular index key will have to be used.

2.3.3 Internal Workings

The associative array data structure has an internal tree. This tree is constructed of two types of metadata blocks: nodes and shortcuts.

A node is an array of slots. Each slot can contain one of four things:

- A NULL pointer, indicating that the slot is empty.
- A pointer to an object (a leaf).
- A pointer to a node at the next level.
- A pointer to a shortcut.

Basic Internal Tree Layout

Ignoring shortcuts for the moment, the nodes form a multilevel tree. The index key space is strictly subdivided by the nodes in the tree and nodes occur on fixed levels. For example:

```
Level: 0               1               2               3
       =============== =============== =============== ===============
                                                       NODE D
                       NODE B          NODE C  +------>+---+
               +------>+---+   +------>+---+   |       | 0 |
       NODE A  |       | 0 |   |       | 0 |   |       +---+
       +---+   |       +---+   |       +---+   |       :   :
       | 0 |   |       :   :   |       :   :   |       +---+
       +---+   |       +---+   |       +---+   |       | f |
       | 1 |---+       | 3 |---+       | 7 |---+       +---+
       +---+           +---+           +---+
       :   :           :   :           | 8 |---+
       +---+           +---+           +---+   |       NODE E
       | e |---+       | f |           :   :   +------>+---+
       +---+   |       +---+           +---+           | 0 |
       | f |   |                       | f |           +---+
       +---+   |                       +---+           :   :
               |       NODE F                          +---+
               +------>+---+                           | f |
                       | 0 |           NODE G          +---+
                       +---+   +------>+---+
                       :   :   |       | 0 |
                       +---+   |       +---+
                       | 6 |---+       :   :
                       +---+           +---+
                       :   :           | f |
                       +---+           +---+
                       | f |
                       +---+
```

In the above example, there are 7 nodes (A-G), each with 16 slots (0-f). Assuming no other meta data nodes in the tree, the key space is divided thusly:

```
KEY PREFIX      NODE
==========      ====
137*            D
138*            E
13[0-69-f]*     C
1[0-24-f]*      B
e6*             G
e[0-57-f]*      F
[02-df]*        A
```

So, for instance, keys with the following example index keys will be found in the appropriate nodes:

```
INDEX KEY       PREFIX  NODE
=============== ======= ====
13694892892489  13      C
13795289025897  137     D
13889dde88793   138     E
138bbb89003093  138     E
1394879524789   12      C
1458952489      1       B
9431809de993ba  -       A
b4542910809cd   -       A
e5284310def98   e       F
e68428974237    e6      G
e7fffcbd443     e       F
f3842239082     -       A
```

To save memory, if a node can hold all the leaves in its portion of keyspace, then the node will have all those leaves in it and will not have any metadata pointers - even if some of those leaves would like to be in the same slot.

A node can contain a heterogeneous mix of leaves and metadata pointers. Metadata pointers must be in the slots that match their subdivisions of key space. The leaves can be in any slot not occupied by a metadata pointer. It is guaranteed that none of the leaves in a node will match a slot occupied by a metadata pointer. If the metadata pointer is there, any leaf whose key matches the metadata key prefix must be in the subtree that the metadata pointer points to.

In the above example list of index keys, node A will contain:

```
SLOT    CONTENT         INDEX KEY (PREFIX)
====    =============== ==================
1       PTR TO NODE B   1*
any     LEAF            9431809de993ba
any     LEAF            b4542910809cd
e       PTR TO NODE F   e*
any     LEAF            f3842239082
```

and node B:

```
3   PTR TO NODE C   13*
any LEAF            1458952489
```

Shortcuts

Shortcuts are metadata records that jump over a piece of keyspace. A shortcut is a replacement for a series of single-occupancy nodes ascending through the levels. Shortcuts exist to save memory and to speed up traversal.

It is possible for the root of the tree to be a shortcut - say, for example, the tree contains at least 17 nodes all with key prefix `1111`. The insertion algorithm will insert a shortcut to skip over the `1111` keyspace in a single bound and get to the fourth level where these actually become different.

Splitting And Collapsing Nodes

Each node has a maximum capacity of 16 leaves and metadata pointers. If the insertion algorithm finds that it is trying to insert a 17th object into a node, that node will be split such that at least two leaves that have a common key segment at that level end up in a separate node rooted on that slot for that common key segment.

If the leaves in a full node and the leaf that is being inserted are sufficiently similar, then a shortcut will be inserted into the tree.

When the number of objects in the subtree rooted at a node falls to 16 or fewer, then the subtree will be collapsed down to a single node - and this will ripple towards the root if possible.

Non-Recursive Iteration

Each node and shortcut contains a back pointer to its parent and the number of slot in that parent that points to it. None-recursive iteration uses these to proceed rootwards through the tree, going to the parent node, slot N + 1 to make sure progress is made without the need for a stack.

The backpointers, however, make simultaneous alteration and iteration tricky.

Simultaneous Alteration And Iteration

There are a number of cases to consider:

1. Simple insert/replace. This involves simply replacing a NULL or old matching leaf pointer with the pointer to the new leaf after a barrier. The metadata blocks don't change otherwise. An old leaf won't be freed until after the RCU grace period.
2. Simple delete. This involves just clearing an old matching leaf. The metadata blocks don't change otherwise. The old leaf won't be freed until after the RCU grace period.
3. Insertion replacing part of a subtree that we haven't yet entered. This may involve replacement of part of that subtree - but that won't affect the iteration as we won't have reached the pointer to it yet and the ancestry blocks are not replaced (the layout of those does not change).
4. Insertion replacing nodes that we're actively processing. This isn't a problem as we've passed the anchoring pointer and won't switch onto the new layout until we follow the back pointers - at which point we've already examined the leaves in the replaced node (we iterate over all the leaves in a node before following any of its metadata pointers).

   We might, however, re-see some leaves that have been split out into a new branch that's in a slot further along than we were at.
5. Insertion replacing nodes that we're processing a dependent branch of. This won't affect us until we follow the back pointers. Similar to (4).
6. Deletion collapsing a branch under us. This doesn't affect us because the back pointers will get us back to the parent of the new node before we could see the new node. The entire collapsed subtree is thrown away unchanged - and will still be rooted on the same slot, so we shouldn't process it a second time as we'll go back to slot + 1.

Note: Under some circumstances, we need to simultaneously change the parent pointer and the parent slot pointer on a node (say, for example, we inserted another node before it and moved it up a level). We cannot do this without locking against a read - so we have to replace that node too.

However, when we're changing a shortcut into a node this isn't a problem as shortcuts only have one slot and so the parent slot number isn't used when traversing backwards over one. This means that it's okay to change the slot number first - provided suitable barriers are used to make sure the parent slot number is read after the back pointer.

Obsolete blocks and leaves are freed up after an RCU grace period has passed, so as long as anyone doing walking or iteration holds the RCU read lock, the old superstructure should not go away on them.

2.4 XArray

Author
Matthew Wilcox

2.4.1 Overview

The XArray is an abstract data type which behaves like a very large array of pointers. It meets many of the same needs as a hash or a conventional resizable array. Unlike a hash, it allows you to sensibly go to the next or previous entry in a cache-efficient manner. In contrast to a resizable array, there is no need to copy data or change MMU mappings in order to grow the array. It is more memory-efficient, parallelisable and cache friendly than a doubly-linked list. It takes advantage of RCU to perform lookups without locking.

The XArray implementation is efficient when the indices used are densely clustered; hashing the object and using the hash as the index will not perform well. The XArray is optimised for small indices, but still has good performance with large indices. If your index can be larger than `ULONG_MAX` then the XArray is not the data type for you. The most important user of the XArray is the page cache.

Normal pointers may be stored in the XArray directly. They must be 4-byte aligned, which is true for any pointer returned from kmalloc() and alloc_page(). It isn't true for arbitrary user-space pointers, nor for function pointers. You can store pointers to statically allocated objects, as long as those objects have an alignment of at least 4.

You can also store integers between 0 and `LONG_MAX` in the XArray. You must first convert it into an entry using xa_mk_value(). When you retrieve an entry from the XArray, you can check whether it is a value entry by calling xa_is_value(), and convert it back to an integer by calling xa_to_value().

Some users want to tag the pointers they store in the XArray. You can call xa_tag_pointer() to create an entry with a tag, xa_untag_pointer() to turn a tagged entry back into an untagged pointer and xa_pointer_tag() to retrieve the tag of an entry. Tagged pointers use the same bits that are used to distinguish value entries from normal pointers, so you must decide whether they want to store value entries or tagged pointers in any particular XArray.

The XArray does not support storing IS_ERR() pointers as some conflict with value entries or internal entries.

An unusual feature of the XArray is the ability to create entries which occupy a range of indices. Once stored to, looking up any index in the range will return the same entry as looking up any other index in the range. Storing to any index will store to all of them. Multi-index entries can be explicitly split into smaller entries, or storing `NULL` into any entry will cause the XArray to forget about the range.

2.4.2 Normal API

Start by initialising an XArray, either with DEFINE_XARRAY() for statically allocated XArrays or xa_init() for dynamically allocated ones. A freshly-initialised XArray contains a `NULL` pointer at every index.

You can then set entries using xa_store() and get entries using xa_load(). xa_store will overwrite any entry with the new entry and return the previous entry stored at that index. You can use xa_erase() instead of calling xa_store() with a `NULL` entry. There is no difference between an entry that has never been stored to, one that has been erased and one that has most recently had `NULL` stored to it.

You can conditionally replace an entry at an index by using xa_cmpxchg(). Like cmpxchg(), it will only succeed if the entry at that index has the 'old' value. It also returns the entry which was at that index; if it returns the same entry which was passed as 'old', then xa_cmpxchg() succeeded.

If you want to only store a new entry to an index if the current entry at that index is `NULL`, you can use xa_insert() which returns `-EBUSY` if the entry is not empty.

You can copy entries out of the XArray into a plain array by calling xa_extract(). Or you can iterate over the present entries in the XArray by calling xa_for_each(), xa_for_each_start() or xa_for_each_range(). You may prefer to use xa_find() or xa_find_after() to move to the next present entry in the XArray.

Calling xa_store_range() stores the same entry in a range of indices. If you do this, some of the other operations will behave in a slightly odd way. For example, marking the entry at one index may result in the entry being marked at some, but not all of the other indices. Storing into one index may result in the entry retrieved by some, but not all of the other indices changing.

Sometimes you need to ensure that a subsequent call to xa_store() will not need to allocate memory. The xa_reserve() function will store a reserved entry at the indicated index. Users of the normal API will see this entry as containing `NULL`. If you do not need to use the reserved entry, you can call xa_release() to remove the unused entry. If another user has stored to the entry in the meantime, xa_release() will do nothing; if instead you want the entry to become `NULL`, you should use xa_erase(). Using xa_insert() on a reserved entry will fail.

If all entries in the array are `NULL`, the xa_empty() function will return `true`.

Finally, you can remove all entries from an XArray by calling xa_destroy(). If the XArray entries are pointers, you may wish to free the entries first. You can do this by iterating over all present entries in the XArray using the xa_for_each() iterator.

Search Marks

Each entry in the array has three bits associated with it called marks. Each mark may be set or cleared independently of the others. You can iterate over marked entries by using the xa_for_each_marked() iterator.

You can enquire whether a mark is set on an entry by using xa_get_mark(). If the entry is not `NULL`, you can set a mark on it by using xa_set_mark() and remove the mark from an entry by calling xa_clear_mark(). You can ask whether any entry in the XArray has a particular mark set by calling xa_marked(). Erasing an entry from the XArray causes all marks associated with that entry to be cleared.

Setting or clearing a mark on any index of a multi-index entry will affect all indices covered by that entry. Querying the mark on any index will return the same result.

There is no way to iterate over entries which are not marked; the data structure does not allow this to be implemented efficiently. There are not currently iterators to search for logical combinations of bits (eg iterate over all entries which have both `XA_MARK_1` and `XA_MARK_2` set, or iterate over all entries which have `XA_MARK_0` or `XA_MARK_2` set). It would be possible to add these if a user arises.

Allocating XArrays

If you use DEFINE_XARRAY_ALLOC() to define the XArray, or initialise it by passing `XA_FLAGS_ALLOC` to xa_init_flags(), the XArray changes to track whether entries are in use or not.

You can call xa_alloc() to store the entry at an unused index in the XArray. If you need to modify the array from interrupt context, you can use xa_alloc_bh() or xa_alloc_irq() to disable interrupts while allocating the ID.

Using xa_store(), xa_cmpxchg() or xa_insert() will also mark the entry as being allocated. Unlike a normal XArray, storing `NULL` will mark the entry as being in use, like xa_reserve(). To free an entry, use xa_erase() (or xa_release() if you only want to free the entry if it's `NULL`).

By default, the lowest free entry is allocated starting from 0. If you want to allocate entries starting at 1, it is more efficient to use DEFINE_XARRAY_ALLOC1() or `XA_FLAGS_ALLOC1`. If you want to allocate IDs up to a maximum, then wrap back around to the lowest free ID, you can use xa_alloc_cyclic().

You cannot use `XA_MARK_0` with an allocating XArray as this mark is used to track whether an entry is free or not. The other marks are available for your use.

Memory allocation

The xa_store(), xa_cmpxchg(), xa_alloc(), xa_reserve() and xa_insert() functions take a gfp_t parameter in case the XArray needs to allocate memory to store this entry. If the entry is being deleted, no memory allocation needs to be performed, and the GFP flags specified will be ignored.

It is possible for no memory to be allocatable, particularly if you pass a restrictive set of GFP flags. In that case, the functions return a special value which can be turned into an errno using xa_err(). If you don't need to know exactly which error occurred, using xa_is_err() is slightly more efficient.

Locking

When using the Normal API, you do not have to worry about locking. The XArray uses RCU and an internal spinlock to synchronise access:

No lock needed:

- xa_empty()
- xa_marked()

Takes RCU read lock:

- xa_load()
- xa_for_each()
- xa_for_each_start()
- xa_for_each_range()
- xa_find()
- xa_find_after()
- xa_extract()
- xa_get_mark()

Takes xa_lock internally:

- xa_store()
- xa_store_bh()
- xa_store_irq()
- xa_insert()
- xa_insert_bh()
- xa_insert_irq()
- xa_erase()
- xa_erase_bh()
- xa_erase_irq()
- xa_cmpxchg()
- xa_cmpxchg_bh()
- xa_cmpxchg_irq()
- xa_store_range()
- xa_alloc()
- xa_alloc_bh()
- xa_alloc_irq()
- xa_reserve()
- xa_reserve_bh()

- xa_reserve_irq()
- xa_destroy()
- xa_set_mark()
- xa_clear_mark()

Assumes xa_lock held on entry:

- __xa_store()
- __xa_insert()
- __xa_erase()
- __xa_cmpxchg()
- __xa_alloc()
- __xa_set_mark()
- __xa_clear_mark()

If you want to take advantage of the lock to protect the data structures that you are storing in the XArray, you can call xa_lock() before calling xa_load(), then take a reference count on the object you have found before calling xa_unlock(). This will prevent stores from removing the object from the array between looking up the object and incrementing the refcount. You can also use RCU to avoid dereferencing freed memory, but an explanation of that is beyond the scope of this document.

The XArray does not disable interrupts or softirqs while modifying the array. It is safe to read the XArray from interrupt or softirq context as the RCU lock provides enough protection.

If, for example, you want to store entries in the XArray in process context and then erase them in softirq context, you can do that this way:

```
void foo_init(struct foo *foo)
{
    xa_init_flags(&foo->array, XA_FLAGS_LOCK_BH);
}

int foo_store(struct foo *foo, unsigned long index, void *entry)
{
    int err;

    xa_lock_bh(&foo->array);
    err = xa_err(__xa_store(&foo->array, index, entry, GFP_KERNEL));
    if (!err)
        foo->count++;
    xa_unlock_bh(&foo->array);
    return err;
}

/* foo_erase() is only called from softirq context */
void foo_erase(struct foo *foo, unsigned long index)
{
    xa_lock(&foo->array);
```

```
    __xa_erase(&foo->array, index);
    foo->count--;
    xa_unlock(&foo->array);
}
```

If you are going to modify the XArray from interrupt or softirq context, you need to initialise the array using xa_init_flags(), passing `XA_FLAGS_LOCK_IRQ` or `XA_FLAGS_LOCK_BH`.

The above example also shows a common pattern of wanting to extend the coverage of the xa_lock on the store side to protect some statistics associated with the array.

Sharing the XArray with interrupt context is also possible, either using xa_lock_irqsave() in both the interrupt handler and process context, or xa_lock_irq() in process context and xa_lock() in the interrupt handler. Some of the more common patterns have helper functions such as xa_store_bh(), xa_store_irq(), xa_erase_bh(), xa_erase_irq(), xa_cmpxchg_bh() and xa_cmpxchg_irq().

Sometimes you need to protect access to the XArray with a mutex because that lock sits above another mutex in the locking hierarchy. That does not entitle you to use functions like __xa_erase() without taking the xa_lock; the xa_lock is used for lockdep validation and will be used for other purposes in the future.

The __xa_set_mark() and __xa_clear_mark() functions are also available for situations where you look up an entry and want to atomically set or clear a mark. It may be more efficient to use the advanced API in this case, as it will save you from walking the tree twice.

2.4.3 Advanced API

The advanced API offers more flexibility and better performance at the cost of an interface which can be harder to use and has fewer safeguards. No locking is done for you by the advanced API, and you are required to use the xa_lock while modifying the array. You can choose whether to use the xa_lock or the RCU lock while doing read-only operations on the array. You can mix advanced and normal operations on the same array; indeed the normal API is implemented in terms of the advanced API. The advanced API is only available to modules with a GPL-compatible license.

The advanced API is based around the xa_state. This is an opaque data structure which you declare on the stack using the XA_STATE() macro. This macro initialises the xa_state ready to start walking around the XArray. It is used as a cursor to maintain the position in the XArray and let you compose various operations together without having to restart from the top every time. The contents of the xa_state are protected by the rcu_read_lock() or the xas_lock(). If you need to drop whichever of those locks is protecting your state and tree, you must call xas_pause() so that future calls do not rely on the parts of the state which were left unprotected.

The xa_state is also used to store errors. You can call xas_error() to retrieve the error. All operations check whether the xa_state is in an error state before proceeding, so there's no need for you to check for an error after each call; you can make multiple calls in succession and only check at a convenient point. The only errors currently generated by the XArray code itself are `ENOMEM` and `EINVAL`, but it supports arbitrary errors in case you want to call xas_set_err() yourself.

If the xa_state is holding an `ENOMEM` error, calling xas_nomem() will attempt to allocate more memory using the specified gfp flags and cache it in the xa_state for the next attempt. The idea is that you take the xa_lock, attempt the operation and drop the lock. The operation attempts to allocate memory while holding the lock, but it is more likely to fail. Once you have dropped the lock, xas_nomem() can try harder to allocate more memory. It will return `true` if it is worth retrying the operation (i.e. that there was a memory error *and* more memory was allocated). If it has previously allocated memory, and that memory wasn't used, and there is no error (or some error that isn't `ENOMEM`), then it will free the memory previously allocated.

Internal Entries

The XArray reserves some entries for its own purposes. These are never exposed through the normal API, but when using the advanced API, it's possible to see them. Usually the best way to handle them is to pass them to xas_retry(), and retry the operation if it returns `true`.

Name	Test	Usage
Node	xa_is_node()	An XArray node. May be visible when using a multi-index xa_state.
Sibling	xa_is_sibling()	A non-canonical entry for a multi-index entry. The value indicates which slot in this node has the canonical entry.
Retry	xa_is_retry()	This entry is currently being modified by a thread which has the xa_lock. The node containing this entry may be freed at the end of this RCU period. You should restart the lookup from the head of the array.
Zero	xa_is_zero()	Zero entries appear as `NULL` through the Normal API, but occupy an entry in the XArray which can be used to reserve the index for future use. This is used by allocating XArrays for allocated entries which are `NULL`.

Other internal entries may be added in the future. As far as possible, they will be handled by xas_retry().

Additional functionality

The xas_create_range() function allocates all the necessary memory to store every entry in a range. It will set ENOMEM in the xa_state if it cannot allocate memory.

You can use xas_init_marks() to reset the marks on an entry to their default state. This is usually all marks clear, unless the XArray is marked with `XA_FLAGS_TRACK_FREE`, in which case mark 0 is set and all other marks are clear. Replacing one entry with another using xas_store() will not reset the marks on that entry; if you want the marks reset, you should do that explicitly.

The xas_load() will walk the xa_state as close to the entry as it can. If you know the xa_state has already been walked to the entry and need to check that the entry hasn't changed, you can use xas_reload() to save a function call.

If you need to move to a different index in the XArray, call xas_set(). This resets the cursor to the top of the tree, which will generally make the next operation walk the cursor to the desired spot in the tree. If you want to move to the next or previous index, call xas_next() or xas_prev(). Setting the index does not walk the cursor around the array so does not require a lock to be held, while moving to the next or previous index does.

You can search for the next present entry using xas_find(). This is the equivalent of both xa_find() and xa_find_after(); if the cursor has been walked to an entry, then it will find the next entry after the one currently referenced. If not, it will return the entry at the index of the xa_state. Using xas_next_entry() to move to the next present entry instead of xas_find() will save a function call in the majority of cases at the expense of emitting more inline code.

The xas_find_marked() function is similar. If the xa_state has not been walked, it will return the entry at the index of the xa_state, if it is marked. Otherwise, it will return the first marked entry after the entry referenced by the xa_state. The xas_next_marked() function is the equivalent of xas_next_entry().

When iterating over a range of the XArray using xas_for_each() or xas_for_each_marked(), it may be necessary to temporarily stop the iteration. The xas_pause() function exists for this purpose. After you have done the necessary work and wish to resume, the xa_state is in an appropriate state to continue the iteration after the entry you last processed. If you have interrupts disabled while iterating, then it is good manners to pause the iteration and reenable interrupts every `XA_CHECK_SCHED` entries.

The xas_get_mark(), xas_set_mark() and xas_clear_mark() functions require the xa_state cursor to have been moved to the appropriate location in the XArray; they will do nothing if you have called xas_pause() or xas_set() immediately before.

You can call xas_set_update() to have a callback function called each time the XArray updates a node. This is used by the page cache workingset code to maintain its list of nodes which contain only shadow entries.

Multi-Index Entries

The XArray has the ability to tie multiple indices together so that operations on one index affect all indices. For example, storing into any index will change the value of the entry retrieved from any index. Setting or clearing a mark on any index will set or clear the mark on every index that is tied together. The current implementation only allows tying ranges which are aligned powers of two together; eg indices 64-127 may be tied together, but 2-6 may not be. This may save substantial quantities of memory; for example tying 512 entries together will save over 4kB.

You can create a multi-index entry by using XA_STATE_ORDER() or xas_set_order() followed by a call to xas_store(). Calling xas_load() with a multi-index xa_state will walk the xa_state to the right location in the tree, but the return value is not meaningful, potentially being an internal entry or `NULL` even when there is an entry stored within the range. Calling xas_find_conflict() will return the first entry within the range or `NULL` if there are no entries in the range. The xas_for_each_conflict() iterator will iterate over every entry which overlaps the specified range.

If xas_load() encounters a multi-index entry, the xa_index in the xa_state will not be changed. When iterating over an XArray or calling xas_find(), if the initial index is in the middle of a multi-index entry, it will not be altered. Subsequent calls or iterations will move the index to the first index in the range. Each entry will only be returned once, no matter how many indices it occupies.

Using xas_next() or xas_prev() with a multi-index xa_state is not supported. Using either of these functions on a multi-index entry will reveal sibling entries; these should be skipped over by the caller.

Storing `NULL` into any index of a multi-index entry will set the entry at every index to `NULL` and dissolve the tie. A multi-index entry can be split into entries occupying smaller ranges by calling xas_split_alloc() without the xa_lock held, followed by taking the lock and calling xas_split().

2.4.4 Functions and structures

void ***xa_mk_value**(unsigned long v)

Create an XArray entry from an integer.

Parameters

unsigned long v

Value to store in XArray.

Context

Any context.

Return

An entry suitable for storing in the XArray.

unsigned long **xa_to_value**(const void *entry)

Get value stored in an XArray entry.

Parameters

const void *entry

XArray entry.

Context

Any context.

Return

The value stored in the XArray entry.

bool **xa_is_value**(const void *entry)

Determine if an entry is a value.

Parameters

const void *entry

XArray entry.

Context

Any context.

Return

True if the entry is a value, false if it is a pointer.

void ***xa_tag_pointer**(void *p, unsigned long tag)

Create an XArray entry for a tagged pointer.

Parameters

void *p

Plain pointer.

unsigned long tag

Tag value (0, 1 or 3).

Description

If the user of the XArray prefers, they can tag their pointers instead of storing value entries. Three tags are available (0, 1 and 3). These are distinct from the xa_mark_t as they are not replicated up through the array and cannot be searched for.

Context

Any context.

Return

An XArray entry.

void ***xa_untag_pointer**(void *entry)

Turn an XArray entry into a plain pointer.

Parameters

void *entry

XArray entry.

Description

If you have stored a tagged pointer in the XArray, call this function to get the untagged version of the pointer.

Context

Any context.

Return

A pointer.

unsigned int **xa_pointer_tag**(void *entry)

Get the tag stored in an XArray entry.

Parameters

void *entry

XArray entry.

Description

If you have stored a tagged pointer in the XArray, call this function to get the tag of that pointer.

Context

Any context.

Return

A tag.

bool **xa_is_zero**(const void *entry)

Is the entry a zero entry?

Parameters

const void *entry

Entry retrieved from the XArray

Description

The normal API will return NULL as the contents of a slot containing a zero entry. You can only see zero entries by using the advanced API.

Return

`true` if the entry is a zero entry.

bool **xa_is_err**(const void *entry)

Report whether an XArray operation returned an error

Parameters

const void *entry

Result from calling an XArray function

Description

If an XArray operation cannot complete an operation, it will return a special value indicating an error. This function tells you whether an error occurred; xa_err() tells you which error occurred.

Context

Any context.

Return

`true` if the entry indicates an error.

int **xa_err**(void *entry)

Turn an XArray result into an errno.

Parameters

void *entry

Result from calling an XArray function.

Description

If an XArray operation cannot complete an operation, it will return a special pointer value which encodes an errno. This function extracts the errno from the pointer value, or returns 0 if the pointer does not represent an errno.

Context

Any context.

Return

A negative errno or 0.

struct **xa_limit**

Represents a range of IDs.

Definition:

```
struct xa_limit {
    u32 max;
    u32 min;
};
```

Members

max
The maximum ID to allocate (inclusive).

min
The lowest ID to allocate (inclusive).

Description

This structure is used either directly or via the XA_LIMIT() macro to communicate the range of IDs that are valid for allocation. Three common ranges are predefined for you: * xa_limit_32b - [0 - UINT_MAX] * xa_limit_31b - [0 - INT_MAX] * xa_limit_16b - [0 - USHRT_MAX]

struct **xarray**
The anchor of the XArray.

Definition:

```
struct xarray {
    spinlock_t xa_lock;
};
```

Members

xa_lock
Lock that protects the contents of the XArray.

Description

To use the xarray, define it statically or embed it in your data structure. It is a very small data structure, so it does not usually make sense to allocate it separately and keep a pointer to it in your data structure.

You may use the xa_lock to protect your own data structures as well.

DEFINE_XARRAY_FLAGS

`DEFINE_XARRAY_FLAGS (name, flags)`

Define an XArray with custom flags.

Parameters

name
A string that names your XArray.

flags
XA_FLAG values.

Description

This is intended for file scope definitions of XArrays. It declares and initialises an empty XArray with the chosen name and flags. It is equivalent to calling xa_init_flags() on the array, but it does the initialisation at compiletime instead of runtime.

DEFINE_XARRAY

`DEFINE_XARRAY (name)`

Define an XArray.

Parameters

name
A string that names your XArray.

Description

This is intended for file scope definitions of XArrays. It declares and initialises an empty XArray with the chosen name. It is equivalent to calling xa_init() on the array, but it does the initialisation at compiletime instead of runtime.

DEFINE_XARRAY_ALLOC

`DEFINE_XARRAY_ALLOC (name)`

Define an XArray which allocates IDs starting at 0.

Parameters

name
A string that names your XArray.

Description

This is intended for file scope definitions of allocating XArrays. See also DEFINE_XARRAY().

DEFINE_XARRAY_ALLOC1

`DEFINE_XARRAY_ALLOC1 (name)`

Define an XArray which allocates IDs starting at 1.

Parameters

name
A string that names your XArray.

Description

This is intended for file scope definitions of allocating XArrays. See also DEFINE_XARRAY().

void **xa_init_flags**(struct *xarray* *xa, gfp_t flags)

Initialise an empty XArray with flags.

Parameters

struct xarray *xa
XArray.

gfp_t flags
XA_FLAG values.

Description

If you need to initialise an XArray with special flags (eg you need to take the lock from interrupt context), use this function instead of xa_init().

Context

Any context.

void **xa_init**(struct *xarray* *xa)
Initialise an empty XArray.

Parameters

struct xarray *xa
XArray.

Description

An empty XArray is full of NULL entries.

Context

Any context.

bool **xa_empty**(const struct *xarray* *xa)
Determine if an array has any present entries.

Parameters

const struct xarray *xa
XArray.

Context

Any context.

Return

`true` if the array contains only NULL pointers.

bool **xa_marked**(const struct *xarray* *xa, xa_mark_t mark)
Inquire whether any entry in this array has a mark set

Parameters

const struct xarray *xa
Array

xa_mark_t mark
Mark value

Context

Any context.

Return

`true` if any entry has this mark set.

xa_for_each_range

`xa_for_each_range (xa, index, entry, start, last)`

Iterate over a portion of an XArray.

Parameters

xa
XArray.

index
Index of **entry**.

entry
  Entry retrieved from array.

start
  First index to retrieve from array.

last
  Last index to retrieve from array.

Description

During the iteration, **entry** will have the value of the entry stored in **xa** at **index**. You may modify **index** during the iteration if you want to skip or reprocess indices. It is safe to modify the array during the iteration. At the end of the iteration, **entry** will be set to NULL and **index** will have a value less than or equal to max.

xa_for_each_range() is O(n.log(n)) while xas_for_each() is O(n). You have to handle your own locking with xas_for_each(), and if you have to unlock after each iteration, it will also end up being O(n.log(n)). xa_for_each_range() will spin if it hits a retry entry; if you intend to see retry entries, you should use the xas_for_each() iterator instead. The xas_for_each() iterator will expand into more inline code than xa_for_each_range().

Context

Any context. Takes and releases the RCU lock.

xa_for_each_start

`xa_for_each_start (xa, index, entry, start)`

  Iterate over a portion of an XArray.

Parameters

xa
  XArray.

index
  Index of **entry**.

entry
  Entry retrieved from array.

start
  First index to retrieve from array.

Description

During the iteration, **entry** will have the value of the entry stored in **xa** at **index**. You may modify **index** during the iteration if you want to skip or reprocess indices. It is safe to modify the array during the iteration. At the end of the iteration, **entry** will be set to NULL and **index** will have a value less than or equal to max.

xa_for_each_start() is O(n.log(n)) while xas_for_each() is O(n). You have to handle your own locking with xas_for_each(), and if you have to unlock after each iteration, it will also end up being O(n.log(n)). xa_for_each_start() will spin if it hits a retry entry; if you intend to see retry entries, you should use the xas_for_each() iterator instead. The xas_for_each() iterator will expand into more inline code than xa_for_each_start().

Context

Any context. Takes and releases the RCU lock.

xa_for_each

`xa_for_each (xa, index, entry)`

Iterate over present entries in an XArray.

Parameters

xa
XArray.

index
Index of **entry**.

entry
Entry retrieved from array.

Description

During the iteration, **entry** will have the value of the entry stored in **xa** at **index**. You may modify **index** during the iteration if you want to skip or reprocess indices. It is safe to modify the array during the iteration. At the end of the iteration, **entry** will be set to NULL and **index** will have a value less than or equal to max.

xa_for_each() is O(n.log(n)) while xas_for_each() is O(n). You have to handle your own locking with xas_for_each(), and if you have to unlock after each iteration, it will also end up being O(n.log(n)). xa_for_each() will spin if it hits a retry entry; if you intend to see retry entries, you should use the xas_for_each() iterator instead. The xas_for_each() iterator will expand into more inline code than xa_for_each().

Context

Any context. Takes and releases the RCU lock.

xa_for_each_marked

`xa_for_each_marked (xa, index, entry, filter)`

Iterate over marked entries in an XArray.

Parameters

xa
XArray.

index
Index of **entry**.

entry
Entry retrieved from array.

filter
Selection criterion.

Description

During the iteration, **entry** will have the value of the entry stored in **xa** at **index**. The iteration will skip all entries in the array which do not match **filter**. You may modify **index** during the iteration if you want to skip or reprocess indices. It is safe to modify the array during the iteration. At the end of the iteration, **entry** will be set to NULL and **index** will have a value less than or equal to max.

xa_for_each_marked() is O(n.log(n)) while xas_for_each_marked() is O(n). You have to handle your own locking with xas_for_each(), and if you have to unlock after each iteration, it will also end up being O(n.log(n)). xa_for_each_marked() will spin if it hits a retry entry; if you intend to see retry entries, you should use the xas_for_each_marked() iterator instead. The xas_for_each_marked() iterator will expand into more inline code than xa_for_each_marked().

Context

Any context. Takes and releases the RCU lock.

void ***xa_store_bh**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)

Store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

This function is like calling xa_store() except it disables softirqs while holding the array lock.

Context

Any context. Takes and releases the xa_lock while disabling softirqs.

Return

The old entry at this index or xa_err() if an error happened.

void ***xa_store_irq**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)

Store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

This function is like calling xa_store() except it disables interrupts while holding the array lock.

Context

Process context. Takes and releases the xa_lock while disabling interrupts.

Return

The old entry at this index or xa_err() if an error happened.

void ***xa_erase_bh**(struct *xarray* *xa, unsigned long index)

Erase this entry from the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

Description

After this function returns, loading from **index** will return `NULL`. If the index is part of a multi-index entry, all indices will be erased and none of the entries will be part of a multi-index entry.

Context

Any context. Takes and releases the xa_lock while disabling softirqs.

Return

The entry which used to be at this index.

void ***xa_erase_irq**(struct *xarray* *xa, unsigned long index)

Erase this entry from the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

Description

After this function returns, loading from **index** will return `NULL`. If the index is part of a multi-index entry, all indices will be erased and none of the entries will be part of a multi-index entry.

Context

Process context. Takes and releases the xa_lock while disabling interrupts.

Return

The entry which used to be at this index.

void ***xa_cmpxchg**(struct *xarray* *xa, unsigned long index, void *old, void *entry, gfp_t gfp)

Conditionally replace an entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *old
Old value to test against.

void *entry
New value to place in array.

gfp_t gfp
Memory allocation flags.

Description

If the entry at **index** is the same as **old**, replace it with **entry**. If the return value is equal to **old**, then the exchange was successful.

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

The old value at this index or xa_err() if an error happened.

void ***xa_cmpxchg_bh**(struct *xarray* *xa, unsigned long index, void *old, void *entry, gfp_t gfp)
Conditionally replace an entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *old
Old value to test against.

void *entry
New value to place in array.

gfp_t gfp
Memory allocation flags.

Description

This function is like calling xa_cmpxchg() except it disables softirqs while holding the array lock.

Context

Any context. Takes and releases the xa_lock while disabling softirqs. May sleep if the **gfp** flags permit.

Return

The old value at this index or xa_err() if an error happened.

void ***xa_cmpxchg_irq**(struct *xarray* *xa, unsigned long index, void *old, void *entry, gfp_t gfp)
Conditionally replace an entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *old
Old value to test against.

void *entry
New value to place in array.

gfp_t gfp
Memory allocation flags.

Description

This function is like calling xa_cmpxchg() except it disables interrupts while holding the array lock.

Context

Process context. Takes and releases the xa_lock while disabling interrupts. May sleep if the **gfp** flags permit.

Return

The old value at this index or xa_err() if an error happened.

int **xa_insert**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)
Store this entry in the XArray unless another entry is already present.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

Inserting a NULL entry will store a reserved entry (like xa_reserve()) if no entry is present. Inserting will fail if a reserved entry is present, even though loading from this index will return NULL.

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

0 if the store succeeded. -EBUSY if another entry was present. -ENOMEM if memory could not be allocated.

int **xa_insert_bh**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)
Store this entry in the XArray unless another entry is already present.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

Inserting a NULL entry will store a reserved entry (like xa_reserve()) if no entry is present. Inserting will fail if a reserved entry is present, even though loading from this index will return NULL.

Context

Any context. Takes and releases the xa_lock while disabling softirqs. May sleep if the **gfp** flags permit.

Return

0 if the store succeeded. -EBUSY if another entry was present. -ENOMEM if memory could not be allocated.

int **xa_insert_irq**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)
Store this entry in the XArray unless another entry is already present.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

Inserting a NULL entry will store a reserved entry (like xa_reserve()) if no entry is present. Inserting will fail if a reserved entry is present, even though loading from this index will return NULL.

Context

Process context. Takes and releases the xa_lock while disabling interrupts. May sleep if the **gfp** flags permit.

Return

0 if the store succeeded. -EBUSY if another entry was present. -ENOMEM if memory could not be allocated.

int **xa_alloc**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

0 on success, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_alloc_bh**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Takes and releases the xa_lock while disabling softirqs. May sleep if the **gfp** flags permit.

Return

0 on success, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_alloc_irq**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Process context. Takes and releases the xa_lock while disabling interrupts. May sleep if the **gfp** flags permit.

Return

0 on success, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_alloc_cyclic**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, u32 *next, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of allocated ID.

u32 *next
Pointer to next ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**. The search for an empty entry will start at **next** and will wrap around if necessary.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

0 if the allocation succeeded without wrapping. 1 if the allocation succeeded after wrapping, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_alloc_cyclic_bh**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, u32 *next, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of allocated ID.

u32 *next
Pointer to next ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**. The search for an empty entry will start at **next** and will wrap around if necessary.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Takes and releases the xa_lock while disabling softirqs. May sleep if the **gfp** flags permit.

Return

0 if the allocation succeeded without wrapping. 1 if the allocation succeeded after wrapping, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_alloc_cyclic_irq**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, u32 *next, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of allocated ID.

u32 *next
Pointer to next ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**. The search for an empty entry will start at **next** and will wrap around if necessary.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Process context. Takes and releases the xa_lock while disabling interrupts. May sleep if the **gfp** flags permit.

Return

0 if the allocation succeeded without wrapping. 1 if the allocation succeeded after wrapping, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **xa_reserve**(struct *xarray* *xa, unsigned long index, gfp_t gfp)

Reserve this index in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

gfp_t gfp
Memory allocation flags.

Description

Ensures there is somewhere to store an entry at **index** in the array. If there is already something stored at **index**, this function does nothing. If there was nothing there, the entry is marked as reserved. Loading from a reserved entry returns a `NULL` pointer.

If you do not use the entry that you have reserved, call xa_release() or xa_erase() to free any unnecessary memory.

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

0 if the reservation succeeded or -ENOMEM if it failed.

int **xa_reserve_bh**(struct *xarray* *xa, unsigned long index, gfp_t gfp)
Reserve this index in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

gfp_t gfp
Memory allocation flags.

Description

A softirq-disabling version of xa_reserve().

Context

Any context. Takes and releases the xa_lock while disabling softirqs.

Return

0 if the reservation succeeded or -ENOMEM if it failed.

int **xa_reserve_irq**(struct *xarray* *xa, unsigned long index, gfp_t gfp)
Reserve this index in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

gfp_t gfp
Memory allocation flags.

Description

An interrupt-disabling version of xa_reserve().

Context

Process context. Takes and releases the xa_lock while disabling interrupts.

Return

0 if the reservation succeeded or -ENOMEM if it failed.

void **xa_release**(struct *xarray* *xa, unsigned long index)
Release a reserved entry.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

Description

After calling xa_reserve(), you can call this function to release the reservation. If the entry at **index** has been stored to, this function will do nothing.

bool **xa_is_sibling**(const void *entry)
Is the entry a sibling entry?

Parameters

const void *entry
Entry retrieved from the XArray

Return

`true` if the entry is a sibling entry.

bool **xa_is_retry**(const void *entry)
Is the entry a retry entry?

Parameters

const void *entry
Entry retrieved from the XArray

Return

`true` if the entry is a retry entry.

bool **xa_is_advanced**(const void *entry)
Is the entry only permitted for the advanced API?

Parameters

const void *entry
Entry to be stored in the XArray.

Return

`true` if the entry cannot be stored by the normal API.

xa_update_node_t

Typedef: A callback function from the XArray.

Syntax

`void xa_update_node_t (struct xa_node *node)`

Parameters

struct xa_node *node
The node which is being processed

Description

This function is called every time the XArray updates the count of present and value entries in a node. It allows advanced users to maintain the private_list in the node.

Context

The xa_lock is held and interrupts may be disabled. Implementations should not drop the xa_lock, nor re-enable interrupts.

XA_STATE

`XA_STATE (name, array, index)`

Declare an XArray operation state.

Parameters

name
Name of this operation state (usually xas).

array
Array to operate on.

index
Initial index of interest.

Description

Declare and initialise an xa_state on the stack.

XA_STATE_ORDER

`XA_STATE_ORDER (name, array, index, order)`

Declare an XArray operation state.

Parameters

name
Name of this operation state (usually xas).

array
Array to operate on.

index
Initial index of interest.

order
Order of entry.

Description

Declare and initialise an xa_state on the stack. This variant of XA_STATE() allows you to specify the 'order' of the element you want to operate on.`

int **xas_error**(const struct xa_state *xas)

Return an errno stored in the xa_state.

Parameters

const struct xa_state *xas
XArray operation state.

Return

0 if no error has been noted. A negative errno if one has.

void **xas_set_err**(struct xa_state *xas, long err)

Note an error in the xa_state.

Parameters

struct xa_state *xas
XArray operation state.

long err
Negative error number.

Description

Only call this function with a negative **err**; zero or positive errors will probably not behave the way you think they should. If you want to clear the error from an xa_state, use *xas_reset()*.

bool **xas_invalid**(const struct xa_state *xas)

Is the xas in a retry or error state?

Parameters

const struct xa_state *xas
XArray operation state.

Return

`true` if the xas cannot be used for operations.

bool **xas_valid**(const struct xa_state *xas)

Is the xas a valid cursor into the array?

Parameters

const struct xa_state *xas
XArray operation state.

Return

`true` if the xas can be used for operations.

bool **xas_is_node**(const struct xa_state *xas)

Does the xas point to a node?

Parameters

const struct xa_state *xas
XArray operation state.

Return

`true` if the xas currently references a node.

void **xas_reset**(struct xa_state *xas)
Reset an XArray operation state.

Parameters

struct xa_state *xas
XArray operation state.

Description

Resets the error or walk state of the **xas** so future walks of the array will start from the root. Use this if you have dropped the xarray lock and want to reuse the xa_state.

Context

Any context.

bool **xas_retry**(struct xa_state *xas, const void *entry)
Retry the operation if appropriate.

Parameters

struct xa_state *xas
XArray operation state.

const void *entry
Entry from xarray.

Description

The advanced functions may sometimes return an internal entry, such as a retry entry or a zero entry. This function sets up the **xas** to restart the walk from the head of the array if needed.

Context

Any context.

Return

true if the operation needs to be retried.

void ***xas_reload**(struct xa_state *xas)
Refetch an entry from the xarray.

Parameters

struct xa_state *xas
XArray operation state.

Description

Use this function to check that a previously loaded entry still has the same value. This is useful for the lockless pagecache lookup where we walk the array with only the RCU lock to protect us, lock the page, then check that the page hasn't moved since we looked it up.

The caller guarantees that **xas** is still valid. If it may be in an error or restart state, call xas_load() instead.

Return

The entry at this location in the xarray.

void **xas_set**(struct xa_state *xas, unsigned long index)

Set up XArray operation state for a different index.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long index
New index into the XArray.

Description

Move the operation state to refer to a different index. This will have the effect of starting a walk from the top; see xas_next() to move to an adjacent index.

void **xas_advance**(struct xa_state *xas, unsigned long index)

Skip over sibling entries.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long index
Index of last sibling entry.

Description

Move the operation state to refer to the last sibling entry. This is useful for loops that normally want to see sibling entries but sometimes want to skip them. Use xas_set() if you want to move to an index which is not part of this entry.

void **xas_set_order**(struct xa_state *xas, unsigned long index, unsigned int order)

Set up XArray operation state for a multislot entry.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long index
Target of the operation.

unsigned int order
Entry occupies 2^**order** indices.

void **xas_set_update**(struct xa_state *xas, *xa_update_node_t* update)

Set up XArray operation state for a callback.

Parameters

struct xa_state *xas
XArray operation state.

xa_update_node_t update
Function to call when updating a node.

Description

The XArray can notify a caller after it has updated an xa_node. This is advanced functionality and is only needed by the page cache and swap cache.

void *`xas_next_entry`(struct xa_state *xas, unsigned long max)
Advance iterator to next present entry.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long max
Highest index to return.

Description

xas_next_entry() is an inline function to optimise xarray traversal for speed. It is equivalent to calling xas_find(), and will call xas_find() for all the hard cases.

Return

The next present entry after the one currently referred to by **xas**.

void *`xas_next_marked`(struct xa_state *xas, unsigned long max, xa_mark_t mark)
Advance iterator to next marked entry.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long max
Highest index to return.

xa_mark_t mark
Mark to search for.

Description

xas_next_marked() is an inline function to optimise xarray traversal for speed. It is equivalent to calling xas_find_marked(), and will call xas_find_marked() for all the hard cases.

Return

The next marked entry after the one currently referred to by **xas**.

xas_for_each

`xas_for_each (xas, entry, max)`

Iterate over a range of an XArray.

Parameters

xas
XArray operation state.

entry
Entry retrieved from the array.

max
Maximum index to retrieve from array.

Description

The loop body will be executed for each entry present in the xarray between the current xas position and **max**. **entry** will be set to the entry retrieved from the xarray. It is safe to delete entries from the array in the loop body. You should hold either the RCU lock or the xa_lock while iterating. If you need to drop the lock, call xas_pause() first.

xas_for_each_marked

`xas_for_each_marked (xas, entry, max, mark)`

Iterate over a range of an XArray.

Parameters

xas
XArray operation state.

entry
Entry retrieved from the array.

max
Maximum index to retrieve from array.

mark
Mark to search for.

Description

The loop body will be executed for each marked entry in the xarray between the current xas position and **max**. **entry** will be set to the entry retrieved from the xarray. It is safe to delete entries from the array in the loop body. You should hold either the RCU lock or the xa_lock while iterating. If you need to drop the lock, call xas_pause() first.

xas_for_each_conflict

`xas_for_each_conflict (xas, entry)`

Iterate over a range of an XArray.

Parameters

xas
XArray operation state.

entry
Entry retrieved from the array.

Description

The loop body will be executed for each entry in the XArray that lies within the range specified by **xas**. If the loop terminates normally, **entry** will be `NULL`. The user may break out of the loop, which will leave **entry** set to the conflicting entry. The caller may also call xa_set_err() to exit the loop while setting an error to record the reason.

void *`xas_prev`(struct xa_state *xas)
Move iterator to previous index.

Parameters

struct xa_state *xas
XArray operation state.

Description

If the **xas** was in an error state, it will remain in an error state and this function will return `NULL`. If the **xas** has never been walked, it will have the effect of calling xas_load(). Otherwise one will be subtracted from the index and the state will be walked to the correct location in the array for the next operation.

If the iterator was referencing index 0, this function wraps around to `ULONG_MAX`.

Return

The entry at the new index. This may be `NULL` or an internal entry.

void *`xas_next`(struct xa_state *xas)
Move state to next index.

Parameters

struct xa_state *xas
XArray operation state.

Description

If the **xas** was in an error state, it will remain in an error state and this function will return `NULL`. If the **xas** has never been walked, it will have the effect of calling xas_load(). Otherwise one will be added to the index and the state will be walked to the correct location in the array for the next operation.

If the iterator was referencing index `ULONG_MAX`, this function wraps around to 0.

Return

The entry at the new index. This may be `NULL` or an internal entry.

void *`xas_load`(struct xa_state *xas)
Load an entry from the XArray (advanced).

Parameters

struct xa_state *xas
XArray operation state.

Description

Usually walks the **xas** to the appropriate state to load the entry stored at xa_index. However, it will do nothing and return `NULL` if **xas** is in an error state. xas_load() will never expand the tree.

If the xa_state is set up to operate on a multi-index entry, xas_load() may return `NULL` or an internal entry, even if there are entries present within the range specified by **xas**.

Context

Any context. The caller should hold the xa_lock or the RCU lock.

Return

Usually an entry in the XArray, but see description for exceptions.

bool **xas_nomem**(struct xa_state *xas, gfp_t gfp)

Allocate memory if needed.

Parameters

struct xa_state *xas
XArray operation state.

gfp_t gfp
Memory allocation flags.

Description

If we need to add new nodes to the XArray, we try to allocate memory with GFP_NOWAIT while holding the lock, which will usually succeed. If it fails, **xas** is flagged as needing memory to continue. The caller should drop the lock and call xas_nomem(). If xas_nomem() succeeds, the caller should retry the operation.

Forward progress is guaranteed as one node is allocated here and stored in the xa_state where it will be found by xas_alloc(). More nodes will likely be found in the slab allocator, but we do not tie them up here.

Return

true if memory was needed, and was successfully allocated.

void **xas_free_nodes**(struct xa_state *xas, struct xa_node *top)

Free this node and all nodes that it references

Parameters

struct xa_state *xas
Array operation state.

struct xa_node *top
Node to free

Description

This node has been removed from the tree. We must now free it and all of its subnodes. There may be RCU walkers with references into the tree, so we must replace all entries with retry markers.

void **xas_create_range**(struct xa_state *xas)

Ensure that stores to this range will succeed

Parameters

struct xa_state *xas
XArray operation state.

Description

Creates all of the slots in the range covered by **xas**. Sets **xas** to create single-index entries and positions it at the beginning of the range. This is for the benefit of users which have not yet been converted to use multi-index entries.

void *`xas_store`(struct xa_state *xas, void *entry)

Store this entry in the XArray.

Parameters

`struct xa_state *xas`
XArray operation state.

`void *entry`
New entry.

Description

If **xas** is operating on a multi-index entry, the entry returned by this function is essentially meaningless (it may be an internal entry or it may be `NULL`, even if there are non-NULL entries at some of the indices covered by the range). This is not a problem for any current users, and can be changed if needed.

Return

The old entry at this index.

bool `xas_get_mark`(const struct xa_state *xas, xa_mark_t mark)

Returns the state of this mark.

Parameters

`const struct xa_state *xas`
XArray operation state.

`xa_mark_t mark`
Mark number.

Return

true if the mark is set, false if the mark is clear or **xas** is in an error state.

void `xas_set_mark`(const struct xa_state *xas, xa_mark_t mark)

Sets the mark on this entry and its parents.

Parameters

`const struct xa_state *xas`
XArray operation state.

`xa_mark_t mark`
Mark number.

Description

Sets the specified mark on this entry, and walks up the tree setting it on all the ancestor entries. Does nothing if **xas** has not been walked to an entry, or is in an error state.

void `xas_clear_mark`(const struct xa_state *xas, xa_mark_t mark)

Clears the mark on this entry and its parents.

Parameters

`const struct xa_state *xas`
XArray operation state.

xa_mark_t mark
Mark number.

Description

Clears the specified mark on this entry, and walks back to the head attempting to clear it on all the ancestor entries. Does nothing if **xas** has not been walked to an entry, or is in an error state.

void **xas_init_marks**(const struct xa_state *xas)
Initialise all marks for the entry

Parameters

const struct xa_state *xas
Array operations state.

Description

Initialise all marks for the entry specified by **xas**. If we're tracking free entries with a mark, we need to set it on all entries. All other marks are cleared.

This implementation is not as efficient as it could be; we may walk up the tree multiple times.

void **xas_split_alloc**(struct xa_state *xas, void *entry, unsigned int order, gfp_t gfp)
Allocate memory for splitting an entry.

Parameters

struct xa_state *xas
XArray operation state.

void *entry
New entry which will be stored in the array.

unsigned int order
Current entry order.

gfp_t gfp
Memory allocation flags.

Description

This function should be called before calling xas_split(). If necessary, it will allocate new nodes (and fill them with **entry**) to prepare for the upcoming split of an entry of **order** size into entries of the order stored in the **xas**.

Context

May sleep if **gfp** flags permit.

void **xas_split**(struct xa_state *xas, void *entry, unsigned int order)
Split a multi-index entry into smaller entries.

Parameters

struct xa_state *xas
XArray operation state.

void *entry
New entry to store in the array.

unsigned int order
Current entry order.

Description

The size of the new entries is set in **xas**. The value in **entry** is copied to all the replacement entries.

Context

Any context. The caller should hold the xa_lock.

void **xas_pause**(struct xa_state *xas)
Pause a walk to drop a lock.

Parameters

struct xa_state *xas
XArray operation state.

Description

Some users need to pause a walk and drop the lock they're holding in order to yield to a higher priority thread or carry out an operation on an entry. Those users should call this function before they drop the lock. It resets the **xas** to be suitable for the next iteration of the loop after the user has reacquired the lock. If most entries found during a walk require you to call xas_pause(), the xa_for_each() iterator may be more appropriate.

Note that xas_pause() only works for forward iteration. If a user needs to pause a reverse iteration, we will need a xas_pause_rev().

void ***xas_find**(struct xa_state *xas, unsigned long max)
Find the next present entry in the XArray.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long max
Highest index to return.

Description

If the **xas** has not yet been walked to an entry, return the entry which has an index >= xas.xa_index. If it has been walked, the entry currently being pointed at has been processed, and so we move to the next entry.

If no entry is found and the array is smaller than **max**, the iterator is set to the smallest index not yet in the array. This allows **xas** to be immediately passed to xas_store().

Return

The entry, if found, otherwise `NULL`.

void ***xas_find_marked**(struct xa_state *xas, unsigned long max, xa_mark_t mark)
Find the next marked entry in the XArray.

Parameters

struct xa_state *xas
XArray operation state.

unsigned long max
Highest index to return.

xa_mark_t mark
Mark number to search for.

Description

If the **xas** has not yet been walked to an entry, return the marked entry which has an index >= xas.xa_index. If it has been walked, the entry currently being pointed at has been processed, and so we return the first marked entry with an index > xas.xa_index.

If no marked entry is found and the array is smaller than **max**, **xas** is set to the bounds state and xas->xa_index is set to the smallest index not yet in the array. This allows **xas** to be immediately passed to xas_store().

If no entry is found before **max** is reached, **xas** is set to the restart state.

Return

The entry, if found, otherwise `NULL`.

void ***xas_find_conflict**(struct xa_state *xas)
Find the next present entry in a range.

Parameters

struct xa_state *xas
XArray operation state.

Description

The **xas** describes both a range and a position within that range.

Context

Any context. Expects xa_lock to be held.

Return

The next entry in the range covered by **xas** or `NULL`.

void ***xa_load**(struct *xarray* *xa, unsigned long index)
Load an entry from an XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
index into array.

Context

Any context. Takes and releases the RCU lock.

Return

The entry at **index** in **xa**.

void *__xa_erase(struct *xarray* *xa, unsigned long index)

Erase this entry from the XArray while locked.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

Description

After this function returns, loading from **index** will return NULL. If the index is part of a multi-index entry, all indices will be erased and none of the entries will be part of a multi-index entry.

Context

Any context. Expects xa_lock to be held on entry.

Return

The entry which used to be at this index.

void *xa_erase(struct *xarray* *xa, unsigned long index)

Erase this entry from the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

Description

After this function returns, loading from **index** will return NULL. If the index is part of a multi-index entry, all indices will be erased and none of the entries will be part of a multi-index entry.

Context

Any context. Takes and releases the xa_lock.

Return

The entry which used to be at this index.

void *__xa_store(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)

Store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

You must already be holding the xa_lock when calling this function. It will drop the lock if needed to allocate memory, and then reacquire it afterwards.

Context

Any context. Expects xa_lock to be held on entry. May release and reacquire xa_lock if **gfp** flags permit.

Return

The old entry at this index or xa_err() if an error happened.

void ***xa_store**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)
Store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

After this function returns, loads from this index will return **entry**. Storing into an existing multi-index entry updates the entry of every index. The marks associated with **index** are unaffected unless **entry** is `NULL`.

Context

Any context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

The old entry at this index on success, xa_err(-EINVAL) if **entry** cannot be stored in an XArray, or xa_err(-ENOMEM) if memory allocation failed.

void ***__xa_cmpxchg**(struct *xarray* *xa, unsigned long index, void *old, void *entry, gfp_t gfp)
Store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *old
Old value to test against.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

You must already be holding the xa_lock when calling this function. It will drop the lock if needed to allocate memory, and then reacquire it afterwards.

Context

Any context. Expects xa_lock to be held on entry. May release and reacquire xa_lock if **gfp** flags permit.

Return

The old entry at this index or xa_err() if an error happened.

int **__xa_insert**(struct *xarray* *xa, unsigned long index, void *entry, gfp_t gfp)
Store this entry in the XArray if no entry is present.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index into array.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

Inserting a NULL entry will store a reserved entry (like xa_reserve()) if no entry is present. Inserting will fail if a reserved entry is present, even though loading from this index will return NULL.

Context

Any context. Expects xa_lock to be held on entry. May release and reacquire xa_lock if **gfp** flags permit.

Return

0 if the store succeeded. -EBUSY if another entry was present. -ENOMEM if memory could not be allocated.

void ***xa_store_range**(struct *xarray* *xa, unsigned long first, unsigned long last, void *entry, gfp_t gfp)
Store this entry at a range of indices in the XArray.

Parameters

struct xarray *xa
XArray.

unsigned long first
First index to affect.

unsigned long last
Last index to affect.

void *entry
New entry.

gfp_t gfp
Memory allocation flags.

Description

After this function returns, loads from any index between **first** and **last**, inclusive will return **entry**. Storing into an existing multi-index entry updates the entry of every index. The marks associated with **index** are unaffected unless **entry** is NULL.

Context

Process context. Takes and releases the xa_lock. May sleep if the **gfp** flags permit.

Return

NULL on success, xa_err(-EINVAL) if **entry** cannot be stored in an XArray, or xa_err(-ENOMEM) if memory allocation failed.

int **xa_get_order**(struct *xarray* *xa, unsigned long index)
Get the order of an entry.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of the entry.

Return

A number between 0 and 63 indicating the order of the entry.

int **__xa_alloc**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, gfp_t gfp)
Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range for allocated ID.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Expects xa_lock to be held on entry. May release and reacquire xa_lock if **gfp** flags permit.

Return

0 on success, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

int **__xa_alloc_cyclic**(struct *xarray* *xa, u32 *id, void *entry, struct *xa_limit* limit, u32 *next, gfp_t gfp)

Find somewhere to store this entry in the XArray.

Parameters

struct xarray *xa
XArray.

u32 *id
Pointer to ID.

void *entry
New entry.

struct xa_limit limit
Range of allocated ID.

u32 *next
Pointer to next ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Finds an empty entry in **xa** between **limit.min** and **limit.max**, stores the index into the **id** pointer, then stores the entry at that index. A concurrent lookup will not see an uninitialised **id**. The search for an empty entry will start at **next** and will wrap around if necessary.

Must only be operated on an xarray initialized with flag XA_FLAGS_ALLOC set in xa_init_flags().

Context

Any context. Expects xa_lock to be held on entry. May release and reacquire xa_lock if **gfp** flags permit.

Return

0 if the allocation succeeded without wrapping. 1 if the allocation succeeded after wrapping, -ENOMEM if memory could not be allocated or -EBUSY if there are no free entries in **limit**.

void **__xa_set_mark**(struct *xarray* *xa, unsigned long index, xa_mark_t mark)
Set this mark on this entry while locked.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

xa_mark_t mark
Mark number.

Description

Attempting to set a mark on a NULL entry does not succeed.

Context

Any context. Expects xa_lock to be held on entry.

void **__xa_clear_mark**(struct *xarray* *xa, unsigned long index, xa_mark_t mark)
Clear this mark on this entry while locked.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

xa_mark_t mark
Mark number.

Context

Any context. Expects xa_lock to be held on entry.

bool **xa_get_mark**(struct *xarray* *xa, unsigned long index, xa_mark_t mark)
Inquire whether this mark is set on this entry.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

xa_mark_t mark
Mark number.

Description

This function uses the RCU read lock, so the result may be out of date by the time it returns. If you need the result to be stable, use a lock.

Context

Any context. Takes and releases the RCU lock.

Return

True if the entry at **index** has this mark set, false if it doesn't.

void **xa_set_mark**(struct *xarray* *xa, unsigned long index, xa_mark_t mark)
Set this mark on this entry.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

xa_mark_t mark
Mark number.

Description

Attempting to set a mark on a NULL entry does not succeed.

Context

Process context. Takes and releases the xa_lock.

void **xa_clear_mark**(struct *xarray* *xa, unsigned long index, xa_mark_t mark)
Clear this mark on this entry.

Parameters

struct xarray *xa
XArray.

unsigned long index
Index of entry.

xa_mark_t mark
Mark number.

Description

Clearing a mark always succeeds.

Context

Process context. Takes and releases the xa_lock.

void ***xa_find**(struct *xarray* *xa, unsigned long *indexp, unsigned long max, xa_mark_t filter)
Search the XArray for an entry.

Parameters

struct xarray *xa
XArray.

unsigned long *indexp
Pointer to an index.

unsigned long max
Maximum index to search to.

xa_mark_t filter
Selection criterion.

Description

Finds the entry in **xa** which matches the **filter**, and has the lowest index that is at least **indexp** and no more than **max**. If an entry is found, **indexp** is updated to be the index of the entry. This function is protected by the RCU read lock, so it may not find entries which are being simultaneously added. It will not return an XA_RETRY_ENTRY; if you need to see retry entries, use xas_find().

Context

Any context. Takes and releases the RCU lock.

Return

The entry, if found, otherwise NULL.

void ***xa_find_after**(struct *xarray* *xa, unsigned long *indexp, unsigned long max, xa_mark_t filter)

Search the XArray for a present entry.

Parameters

struct xarray *xa
XArray.

unsigned long *indexp
Pointer to an index.

unsigned long max
Maximum index to search to.

xa_mark_t filter
Selection criterion.

Description

Finds the entry in **xa** which matches the **filter** and has the lowest index that is above **indexp** and no more than **max**. If an entry is found, **indexp** is updated to be the index of the entry. This function is protected by the RCU read lock, so it may miss entries which are being simultaneously added. It will not return an XA_RETRY_ENTRY; if you need to see retry entries, use xas_find().

Context

Any context. Takes and releases the RCU lock.

Return

The pointer, if found, otherwise NULL.

unsigned int **xa_extract**(struct *xarray* *xa, void **dst, unsigned long start, unsigned long max, unsigned int n, xa_mark_t filter)

Copy selected entries from the XArray into a normal array.

Parameters

struct xarray *xa
The source XArray to copy from.

void **dst
The buffer to copy entries into.

unsigned long start
The first index in the XArray eligible to be selected.

unsigned long max
The last index in the XArray eligible to be selected.

unsigned int n
The maximum number of entries to copy.

xa_mark_t filter
Selection criterion.

Description

Copies up to **n** entries that match **filter** from the XArray. The copied entries will have indices between **start** and **max**, inclusive.

The **filter** may be an XArray mark value, in which case entries which are marked with that mark will be copied. It may also be `XA_PRESENT`, in which case all entries which are not `NULL` will be copied.

The entries returned may not represent a snapshot of the XArray at a moment in time. For example, if another thread stores to index 5, then index 10, calling xa_extract() may return the old contents of index 5 and the new contents of index 10. Indices not modified while this function is running will not be skipped.

If you need stronger guarantees, holding the xa_lock across calls to this function will prevent concurrent modification.

Context

Any context. Takes and releases the RCU lock.

Return

The number of entries copied.

void **xa_delete_node**(struct xa_node *node, *xa_update_node_t* update)
Private interface for workingset code.

Parameters

struct xa_node *node
Node to be removed from the tree.

xa_update_node_t update
Function to call to update ancestor nodes.

Context

xa_lock must be held on entry and will not be released.

void **xa_destroy**(struct *xarray* *xa)
Free all internal data structures.

Parameters

struct xarray *xa
XArray.

Description

After calling this function, the XArray is empty and has freed all memory allocated for its internal data structures. You are responsible for freeing the objects referenced by the XArray.

Context

Any context. Takes and releases the xa_lock, interrupt-safe.

2.5 Maple Tree

Author
Liam R. Howlett

2.5.1 Overview

The Maple Tree is a B-Tree data type which is optimized for storing non-overlapping ranges, including ranges of size 1. The tree was designed to be simple to use and does not require a user written search method. It supports iterating over a range of entries and going to the previous or next entry in a cache-efficient manner. The tree can also be put into an RCU-safe mode of operation which allows reading and writing concurrently. Writers must synchronize on a lock, which can be the default spinlock, or the user can set the lock to an external lock of a different type.

The Maple Tree maintains a small memory footprint and was designed to use modern processor cache efficiently. The majority of the users will be able to use the normal API. An *Advanced API* exists for more complex scenarios. The most important usage of the Maple Tree is the tracking of the virtual memory areas.

The Maple Tree can store values between `0` and `ULONG_MAX`. The Maple Tree reserves values with the bottom two bits set to '10' which are below 4096 (ie 2, 6, 10 .. 4094) for internal use. If the entries may use reserved entries then the users can convert the entries using xa_mk_value() and convert them back by calling xa_to_value(). If the user needs to use a reserved value, then the user can convert the value when using the *Advanced API*, but are blocked by the normal API.

The Maple Tree can also be configured to support searching for a gap of a given size (or larger).

Pre-allocating of nodes is also supported using the *Advanced API*. This is useful for users who must guarantee a successful store operation within a given code segment when allocating cannot be done. Allocations of nodes are relatively small at around 256 bytes.

2.5.2 Normal API

Start by initialising a maple tree, either with DEFINE_MTREE() for statically allocated maple trees or *`mt_init()`* for dynamically allocated ones. A freshly-initialised maple tree contains a `NULL` pointer for the range `0` - `ULONG_MAX`. There are currently two types of maple trees supported: the allocation tree and the regular tree. The regular tree has a higher branching factor for internal nodes. The allocation tree has a lower branching factor but allows the user to search for a gap of a given size or larger from either `0` upwards or `ULONG_MAX` down. An allocation tree can be used by passing in the `MT_FLAGS_ALLOC_RANGE` flag when initialising the tree.

You can then set entries using *mtree_store()* or *mtree_store_range()*. *mtree_store()* will overwrite any entry with the new entry and return 0 on success or an error code otherwise. *mtree_store_range()* works in the same way but takes a range. *mtree_load()* is used to retrieve the entry stored at a given index. You can use *mtree_erase()* to erase an entire range by only knowing one value within that range, or *mtree_store()* call with an entry of NULL may be used to partially erase a range or many ranges at once.

If you want to only store a new entry to a range (or index) if that range is currently `NULL`, you can use *mtree_insert_range()* or *mtree_insert()* which return -EEXIST if the range is not empty.

You can search for an entry from an index upwards by using *mt_find()*.

You can walk each entry within a range by calling *mt_for_each()*. You must provide a temporary variable to store a cursor. If you want to walk each element of the tree then `0` and `ULONG_MAX` may be used as the range. If the caller is going to hold the lock for the duration of the walk then it is worth looking at the *mas_for_each()* API in the *Advanced API* section.

Sometimes it is necessary to ensure the next call to store to a maple tree does not allocate memory, please see *Advanced API* for this use case.

Finally, you can remove all entries from a maple tree by calling *mtree_destroy()*. If the maple tree entries are pointers, you may wish to free the entries first.

Allocating Nodes

The allocations are handled by the internal tree code. See *Advanced Allocating Nodes* for other options.

Locking

You do not have to worry about locking. See *Advanced Locking* for other options.

The Maple Tree uses RCU and an internal spinlock to synchronise access:

Takes RCU read lock:

- *mtree_load()*
- *mt_find()*
- *mt_for_each()*
- *mt_next()*
- *mt_prev()*

Takes ma_lock internally:

- *mtree_store()*
- *mtree_store_range()*
- *mtree_insert()*
- *mtree_insert_range()*
- *mtree_erase()*

- *mtree_destroy()*
- *mt_set_in_rcu()*
- *mt_clear_in_rcu()*

If you want to take advantage of the internal lock to protect the data structures that you are storing in the Maple Tree, you can call mtree_lock() before calling *mtree_load()*, then take a reference count on the object you have found before calling mtree_unlock(). This will prevent stores from removing the object from the tree between looking up the object and incrementing the refcount. You can also use RCU to avoid dereferencing freed memory, but an explanation of that is beyond the scope of this document.

2.5.3 Advanced API

The advanced API offers more flexibility and better performance at the cost of an interface which can be harder to use and has fewer safeguards. You must take care of your own locking while using the advanced API. You can use the ma_lock, RCU or an external lock for protection. You can mix advanced and normal operations on the same array, as long as the locking is compatible. The *Normal API* is implemented in terms of the advanced API.

The advanced API is based around the ma_state, this is where the 'mas' prefix originates. The ma_state struct keeps track of tree operations to make life easier for both internal and external tree users.

Initialising the maple tree is the same as in the *Normal API*. Please see above.

The maple state keeps track of the range start and end in mas->index and mas->last, respectively.

mas_walk() will walk the tree to the location of mas->index and set the mas->index and mas->last according to the range for the entry.

You can set entries using *mas_store()*. *mas_store()* will overwrite any entry with the new entry and return the first existing entry that is overwritten. The range is passed in as members of the maple state: index and last.

You can use *mas_erase()* to erase an entire range by setting index and last of the maple state to the desired range to erase. This will erase the first range that is found in that range, set the maple state index and last as the range that was erased and return the entry that existed at that location.

You can walk each entry within a range by using *mas_for_each()*. If you want to walk each element of the tree then `0` and `ULONG_MAX` may be used as the range. If the lock needs to be periodically dropped, see the locking section *mas_pause()*.

Using a maple state allows *mas_next()* and *mas_prev()* to function as if the tree was a linked list. With such a high branching factor the amortized performance penalty is outweighed by cache optimization. *mas_next()* will return the next entry which occurs after the entry at index. *mas_prev()* will return the previous entry which occurs before the entry at index.

mas_find() will find the first entry which exists at or above index on the first call, and the next entry from every subsequent calls.

mas_find_rev() will find the fist entry which exists at or below the last on the first call, and the previous entry from every subsequent calls.

If the user needs to yield the lock during an operation, then the maple state must be paused using *mas_pause()*.

There are a few extra interfaces provided when using an allocation tree. If you wish to search for a gap within a range, then mas_empty_area() or mas_empty_area_rev() can be used. mas_empty_area() searches for a gap starting at the lowest index given up to the maximum of the range. mas_empty_area_rev() searches for a gap starting at the highest index given and continues downward to the lower bound of the range.

Advanced Allocating Nodes

Allocations are usually handled internally to the tree, however if allocations need to occur before a write occurs then calling mas_expected_entries() will allocate the worst-case number of needed nodes to insert the provided number of ranges. This also causes the tree to enter mass insertion mode. Once insertions are complete calling mas_destroy() on the maple state will free the unused allocations.

Advanced Locking

The maple tree uses a spinlock by default, but external locks can be used for tree updates as well. To use an external lock, the tree must be initialized with the `MT_FLAGS_LOCK_EXTERN flag`, this is usually done with the *MTREE_INIT_EXT()* #define, which takes an external lock as an argument.

2.5.4 Functions and structures

Maple tree flags

- MT_FLAGS_ALLOC_RANGE - Track gaps in this tree
- MT_FLAGS_USE_RCU - Operate in RCU mode
- MT_FLAGS_HEIGHT_OFFSET - The position of the tree height in the flags
- MT_FLAGS_HEIGHT_MASK - The mask for the maple tree height value
- MT_FLAGS_LOCK_MASK - How the mt_lock is used
- MT_FLAGS_LOCK_IRQ - Acquired irq-safe
- MT_FLAGS_LOCK_BH - Acquired bh-safe
- MT_FLAGS_LOCK_EXTERN - mt_lock is not used

MAPLE_HEIGHT_MAX The largest height that can be stored

MTREE_INIT

`MTREE_INIT (name, __flags)`

Initialize a maple tree

Parameters

name
The maple tree name

__flags
The maple tree flags

MTREE_INIT_EXT

`MTREE_INIT_EXT (name, __flags, __lock)`

Initialize a maple tree with an external lock.

Parameters

name
The tree name

__flags
The maple tree flags

__lock
The external lock

bool **mtree_empty**(const struct maple_tree *mt)

Determine if a tree has any present entries.

Parameters

const struct maple_tree *mt
Maple Tree.

Context

Any context.

Return

`true` if the tree contains only NULL pointers.

void **mas_reset**(struct ma_state *mas)

Reset a Maple Tree operation state.

Parameters

struct ma_state *mas
Maple Tree operation state.

Description

Resets the error or walk state of the **mas** so future walks of the array will start from the root. Use this if you have dropped the lock and want to reuse the ma_state.

Context

Any context.

mas_for_each

`mas_for_each (__mas, __entry, __max)`

Iterate over a range of the maple tree.

Parameters

__mas
Maple Tree operation state (maple_state)

__entry
Entry retrieved from the tree

__max
maximum index to retrieve from the tree

Description

When returned, mas->index and mas->last will hold the entire range for the entry.

Note

may return the zero entry.

void **__mas_set_range**(struct ma_state *mas, unsigned long start, unsigned long last)
Set up Maple Tree operation state to a sub-range of the current location.

Parameters

struct ma_state *mas
Maple Tree operation state.

unsigned long start
New start of range in the Maple Tree.

unsigned long last
New end of range in the Maple Tree.

Description

set the internal maple state values to a sub-range. Please use *mas_set_range()* if you do not know where you are in the tree.

void **mas_set_range**(struct ma_state *mas, unsigned long start, unsigned long last)
Set up Maple Tree operation state for a different index.

Parameters

struct ma_state *mas
Maple Tree operation state.

unsigned long start
New start of range in the Maple Tree.

unsigned long last
New end of range in the Maple Tree.

Description

Move the operation state to refer to a different range. This will have the effect of starting a walk from the top; see *mas_next()* to move to an adjacent index.

void **mas_set**(struct ma_state *mas, unsigned long index)
Set up Maple Tree operation state for a different index.

Parameters

struct ma_state *mas
Maple Tree operation state.

unsigned long index
New index into the Maple Tree.

Description

Move the operation state to refer to a different index. This will have the effect of starting a walk from the top; see *mas_next()* to move to an adjacent index.

void **mt_init_flags**(struct maple_tree *mt, unsigned int flags)

Initialise an empty maple tree with flags.

Parameters

struct maple_tree *mt
Maple Tree

unsigned int flags
maple tree flags.

Description

If you need to initialise a Maple Tree with special flags (eg, an allocation tree), use this function.

Context

Any context.

void **mt_init**(struct maple_tree *mt)

Initialise an empty maple tree.

Parameters

struct maple_tree *mt
Maple Tree

Description

An empty Maple Tree.

Context

Any context.

void **mt_clear_in_rcu**(struct maple_tree *mt)

Switch the tree to non-RCU mode.

Parameters

struct maple_tree *mt
The Maple Tree

void **mt_set_in_rcu**(struct maple_tree *mt)

Switch the tree to RCU safe mode.

Parameters

struct maple_tree *mt
The Maple Tree

mt_for_each

`mt_for_each (__tree, __entry, __index, __max)`

Iterate over each entry starting at index until max.

Parameters

__tree
The Maple Tree

__entry
The current entry

__index
The index to start the search from. Subsequently used as iterator.

__max
The maximum limit for **index**

Description

This iterator skips all entries, which resolve to a NULL pointer, e.g. entries which has been reserved with XA_ZERO_ENTRY.

void ***mas_insert**(struct ma_state *mas, void *entry)
Internal call to insert a value

Parameters

struct ma_state *mas
The maple state

void *entry
The entry to store

Return

`NULL` or the contents that already exists at the requested index otherwise. The maple state needs to be checked for error conditions.

void ***mas_walk**(struct ma_state *mas)
Search for **mas->index** in the tree.

Parameters

struct ma_state *mas
The maple state.

Description

mas->index and mas->last will be set to the range if there is a value. If mas->node is MAS_NONE, reset to MAS_START.

Return

the entry at the location or `NULL`.

void __rcu ****mte_dead_walk**(struct maple_enode **enode, unsigned char offset)
Walk down a dead tree to just before the leaves

Parameters

struct maple_enode **enode
The maple encoded node

unsigned char offset
The starting offset

Note

This can only be used from the RCU callback context.

void **mt_free_walk**(struct rcu_head *head)

Walk & free a tree in the RCU callback context

Parameters

struct rcu_head *head
The RCU head that's within the node.

Note

This can only be used from the RCU callback context.

void ***mas_store**(struct ma_state *mas, void *entry)

Store an **entry**.

Parameters

struct ma_state *mas
The maple state.

void *entry
The entry to store.

Description

The **mas->index** and **mas->last** is used to set the range for the **entry**.

Note

The **mas** should have pre-allocated entries to ensure there is memory to store the entry. Please see mas_expected_entries()/mas_destroy() for more details.

Return

the first entry between mas->index and mas->last or `NULL`.

int **mas_store_gfp**(struct ma_state *mas, void *entry, gfp_t gfp)

Store a value into the tree.

Parameters

struct ma_state *mas
The maple state

void *entry
The entry to store

gfp_t gfp
The GFP_FLAGS to use for allocations if necessary.

Return

0 on success, -EINVAL on invalid request, -ENOMEM if memory could not be allocated.

void **mas_store_prealloc**(struct ma_state *mas, void *entry)

Store a value into the tree using memory preallocated in the maple state.

Parameters

struct ma_state *mas
: The maple state

void *entry
: The entry to store.

int **mas_preallocate**(struct ma_state *mas, void *entry, gfp_t gfp)

Preallocate enough nodes for a store operation

Parameters

struct ma_state *mas
: The maple state

void *entry
: The entry that will be stored

gfp_t gfp
: The GFP_FLAGS to use for allocations.

Return

0 on success, -ENOMEM if memory could not be allocated.

void ***mas_next**(struct ma_state *mas, unsigned long max)

Get the next entry.

Parameters

struct ma_state *mas
: The maple state

unsigned long max
: The maximum index to check.

Description

Returns the next entry after **mas->index**. Must hold rcu_read_lock or the write lock. Can return the zero entry.

Return

The next entry or `NULL`

void ***mas_next_range**(struct ma_state *mas, unsigned long max)

Advance the maple state to the next range

Parameters

struct ma_state *mas
: The maple state

unsigned long max
: The maximum index to check.

Description

Sets **mas->index** and **mas->last** to the range. Must hold rcu_read_lock or the write lock. Can return the zero entry.

Return

The next entry or `NULL`

void *`mt_next`(struct maple_tree *mt, unsigned long index, unsigned long max)
get the next value in the maple tree

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The start index

unsigned long max
The maximum index to check

Description

Takes RCU read lock internally to protect the search, which does not protect the returned pointer after dropping RCU read lock. See also: *Maple Tree*

Return

The entry higher than **index** or `NULL` if nothing is found.

void *`mas_prev`(struct ma_state *mas, unsigned long min)
Get the previous entry

Parameters

struct ma_state *mas
The maple state

unsigned long min
The minimum value to check.

Description

Must hold rcu_read_lock or the write lock. Will reset mas to MAS_START if the node is MAS_NONE. Will stop on not searchable nodes.

Return

the previous value or `NULL`.

void *`mas_prev_range`(struct ma_state *mas, unsigned long min)
Advance to the previous range

Parameters

struct ma_state *mas
The maple state

unsigned long min
The minimum value to check.

Description

Sets **mas->index** and **mas->last** to the range. Must hold rcu_read_lock or the write lock. Will reset mas to MAS_START if the node is MAS_NONE. Will stop on not searchable nodes.

Return

the previous value or `NULL`.

void ***mt_prev**(struct maple_tree *mt, unsigned long index, unsigned long min)

get the previous value in the maple tree

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The start index

unsigned long min
The minimum index to check

Description

Takes RCU read lock internally to protect the search, which does not protect the returned pointer after dropping RCU read lock. See also: *Maple Tree*

Return

The entry before **index** or NULL if nothing is found.

void **mas_pause**(struct ma_state *mas)

Pause a mas_find/mas_for_each to drop the lock.

Parameters

struct ma_state *mas
The maple state to pause

Description

Some users need to pause a walk and drop the lock they're holding in order to yield to a higher priority thread or carry out an operation on an entry. Those users should call this function before they drop the lock. It resets the **mas** to be suitable for the next iteration of the loop after the user has reacquired the lock. If most entries found during a walk require you to call *mas_pause()*, the *mt_for_each()* iterator may be more appropriate.

bool **mas_find_setup**(struct ma_state *mas, unsigned long max, void **entry)

Internal function to set up mas_find*().

Parameters

struct ma_state *mas
The maple state

unsigned long max
The maximum index

void **entry
Pointer to the entry

Return

True if entry is the answer, false otherwise.

void ***mas_find**(struct ma_state *mas, unsigned long max)

On the first call, find the entry at or after mas->index up to max. Otherwise, find the entry after mas->index.

Parameters

struct ma_state *mas
The maple state

unsigned long max
The maximum value to check.

Description

Must hold rcu_read_lock or the write lock. If an entry exists, last and index are updated accordingly. May set **mas->node** to MAS_NONE.

Return

The entry or NULL.

void *__mas_find_range__(struct ma_state *mas, unsigned long max)
On the first call, find the entry at or after mas->index up to max. Otherwise, advance to the next slot mas->index.

Parameters

struct ma_state *mas
The maple state

unsigned long max
The maximum value to check.

Description

Must hold rcu_read_lock or the write lock. If an entry exists, last and index are updated accordingly. May set **mas->node** to MAS_NONE.

Return

The entry or NULL.

bool **mas_find_rev_setup**(struct ma_state *mas, unsigned long min, void **entry)
Internal function to set up mas_find_*_rev()

Parameters

struct ma_state *mas
The maple state

unsigned long min
The minimum index

void **entry
Pointer to the entry

Return

True if entry is the answer, false otherwise.

void ***mas_find_rev**(struct ma_state *mas, unsigned long min)
On the first call, find the first non-null entry at or below mas->index down to min. Otherwise find the first non-null entry below mas->index down to min.

Parameters

struct ma_state *mas
    The maple state

unsigned long min
    The minimum value to check.

Description

Must hold rcu_read_lock or the write lock. If an entry exists, last and index are updated accordingly. May set **mas->node** to MAS_NONE.

Return

The entry or `NULL`.

void ***mas_find_range_rev**(struct ma_state *mas, unsigned long min)
    On the first call, find the first non-null entry at or below mas->index down to `min`. Otherwise advance to the previous slot after mas->index down to `min`.

Parameters

struct ma_state *mas
    The maple state

unsigned long min
    The minimum value to check.

Description

Must hold rcu_read_lock or the write lock. If an entry exists, last and index are updated accordingly. May set **mas->node** to MAS_NONE.

Return

The entry or `NULL`.

void ***mas_erase**(struct ma_state *mas)
    Find the range in which index resides and erase the entire range.

Parameters

struct ma_state *mas
    The maple state

Description

Must hold the write lock. Searches for **mas->index**, sets **mas->index** and **mas->last** to the range and erases that range.

Return

the entry that was erased or `NULL`, **mas->index** and **mas->last** are updated.

bool **mas_nomem**(struct ma_state *mas, gfp_t gfp)
    Check if there was an error allocating and do the allocation if necessary If there are allocations, then free them.

Parameters

struct ma_state *mas
    The maple state

gfp_t gfp
The GFP_FLAGS to use for allocations

Return

true on allocation, false otherwise.

void ***mtree_load**(struct maple_tree *mt, unsigned long index)
Load a value stored in a maple tree

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The index to load

Return

the entry or NULL

int **mtree_store_range**(struct maple_tree *mt, unsigned long index, unsigned long last, void *entry, gfp_t gfp)
Store an entry at a given range.

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The start of the range

unsigned long last
The end of the range

void *entry
The entry to store

gfp_t gfp
The GFP_FLAGS to use for allocations

Return

0 on success, -EINVAL on invalid request, -ENOMEM if memory could not be allocated.

int **mtree_store**(struct maple_tree *mt, unsigned long index, void *entry, gfp_t gfp)
Store an entry at a given index.

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The index to store the value

void *entry
The entry to store

gfp_t gfp
The GFP_FLAGS to use for allocations

Return

0 on success, -EINVAL on invalid request, -ENOMEM if memory could not be allocated.

int **mtree_insert_range**(struct maple_tree *mt, unsigned long first, unsigned long last, void *entry, gfp_t gfp)

Insert an entry at a given range if there is no value.

Parameters

struct maple_tree *mt
The maple tree

unsigned long first
The start of the range

unsigned long last
The end of the range

void *entry
The entry to store

gfp_t gfp
The GFP_FLAGS to use for allocations.

Return

0 on success, -EEXISTS if the range is occupied, -EINVAL on invalid request, -ENOMEM if memory could not be allocated.

int **mtree_insert**(struct maple_tree *mt, unsigned long index, void *entry, gfp_t gfp)

Insert an entry at a given index if there is no value.

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
The index to store the value

void *entry
The entry to store

gfp_t gfp
The GFP_FLAGS to use for allocations.

Return

0 on success, -EEXISTS if the range is occupied, -EINVAL on invalid request, -ENOMEM if memory could not be allocated.

void ***mtree_erase**(struct maple_tree *mt, unsigned long index)

Find an index and erase the entire range.

Parameters

struct maple_tree *mt
The maple tree

unsigned long index
: The index to erase

Description

Erasing is the same as a walk to an entry then a store of a NULL to that ENTIRE range. In fact, it is implemented as such using the advanced API.

Return

The entry stored at the **index** or `NULL`

void **`__mt_destroy`**(struct maple_tree *mt)
: Walk and free all nodes of a locked maple tree.

Parameters

struct maple_tree *mt
: The maple tree

Note

Does not handle locking.

void **`mtree_destroy`**(struct maple_tree *mt)
: Destroy a maple tree

Parameters

struct maple_tree *mt
: The maple tree

Description

Frees all resources used by the tree. Handles locking.

void ***`mt_find`**(struct maple_tree *mt, unsigned long *index, unsigned long max)
: Search from the start up until an entry is found.

Parameters

struct maple_tree *mt
: The maple tree

unsigned long *index
: Pointer which contains the start location of the search

unsigned long max
: The maximum value of the search range

Description

Takes RCU read lock internally to protect the search, which does not protect the returned pointer after dropping RCU read lock. See also: *Maple Tree*

In case that an entry is found **index** is updated to point to the next possible entry independent whether the found entry is occupying a single index or a range if indices.

Return

The entry at or after the **index** or `NULL`

void ***mt_find_after**(struct maple_tree *mt, unsigned long *index, unsigned long max)
Search from the start up until an entry is found.

Parameters

struct maple_tree *mt
The maple tree

unsigned long *index
Pointer which contains the start location of the search

unsigned long max
The maximum value to check

Description

Same as *mt_find()* except that it checks **index** for 0 before searching. If **index** == 0, the search is aborted. This covers a wrap around of **index** to 0 in an iterator loop.

Return

The entry at or after the **index** or `NULL`

2.6 ID Allocation

Author
Matthew Wilcox

2.6.1 Overview

A common problem to solve is allocating identifiers (IDs); generally small numbers which identify a thing. Examples include file descriptors, process IDs, packet identifiers in networking protocols, SCSI tags and device instance numbers. The IDR and the IDA provide a reasonable solution to the problem to avoid everybody inventing their own. The IDR provides the ability to map an ID to a pointer, while the IDA provides only ID allocation, and as a result is much more memory-efficient.

The IDR interface is deprecated; please use the *XArray* instead.

2.6.2 IDR usage

Start by initialising an IDR, either with DEFINE_IDR() for statically allocated IDRs or idr_init() for dynamically allocated IDRs.

You can call idr_alloc() to allocate an unused ID. Look up the pointer you associated with the ID by calling idr_find() and free the ID by calling idr_remove().

If you need to change the pointer associated with an ID, you can call idr_replace(). One common reason to do this is to reserve an ID by passing a `NULL` pointer to the allocation function; initialise the object with the reserved ID and finally insert the initialised object into the IDR.

Some users need to allocate IDs larger than `INT_MAX`. So far all of these users have been content with a `UINT_MAX` limit, and they use idr_alloc_u32(). If you need IDs that will not fit in a u32, we will work with you to address your needs.

If you need to allocate IDs sequentially, you can use idr_alloc_cyclic(). The IDR becomes less efficient when dealing with larger IDs, so using this function comes at a slight cost.

To perform an action on all pointers used by the IDR, you can either use the callback-based idr_for_each() or the iterator-style idr_for_each_entry(). You may need to use idr_for_each_entry_continue() to continue an iteration. You can also use idr_get_next() if the iterator doesn't fit your needs.

When you have finished using an IDR, you can call idr_destroy() to release the memory used by the IDR. This will not free the objects pointed to from the IDR; if you want to do that, use one of the iterators to do it.

You can use idr_is_empty() to find out whether there are any IDs currently allocated.

If you need to take a lock while allocating a new ID from the IDR, you may need to pass a restrictive set of GFP flags, which can lead to the IDR being unable to allocate memory. To work around this, you can call idr_preload() before taking the lock, and then idr_preload_end() after the allocation.

idr synchronization (stolen from radix-tree.h)

idr_find() is able to be called locklessly, using RCU. The caller must ensure calls to this function are made within rcu_read_lock() regions. Other readers (lock-free or otherwise) and modifications may be running concurrently.

It is still required that the caller manage the synchronization and lifetimes of the items. So if RCU lock-free lookups are used, typically this would mean that the items have their own locks, or are amenable to lock-free access; and that the items are freed by RCU (or only freed after having been deleted from the idr tree *and* a synchronize_rcu() grace period).

2.6.3 IDA usage

The IDA is an ID allocator which does not provide the ability to associate an ID with a pointer. As such, it only needs to store one bit per ID, and so is more space efficient than an IDR. To use an IDA, define it using DEFINE_IDA() (or embed a `struct ida` in a data structure, then initialise it using ida_init()). To allocate a new ID, call *ida_alloc()*, *ida_alloc_min()*, *ida_alloc_max()* or *ida_alloc_range()*. To free an ID, call *ida_free()*.

ida_destroy() can be used to dispose of an IDA without needing to free the individual IDs in it. You can use ida_is_empty() to find out whether the IDA has any IDs currently allocated.

The IDA handles its own locking. It is safe to call any of the IDA functions without synchronisation in your code.

IDs are currently limited to the range [0-INT_MAX]. If this is an awkward limitation, it should be quite straightforward to raise the maximum.

2.6.4 Functions and structures

IDR_INIT

`IDR_INIT (name)`

Initialise an IDR.

Parameters

name
Name of IDR.

Description

A freshly-initialised IDR contains no IDs.

DEFINE_IDR

`DEFINE_IDR (name)`

Define a statically-allocated IDR.

Parameters

name
Name of IDR.

Description

An IDR defined using this macro is ready for use with no additional initialisation required. It contains no IDs.

unsigned int **idr_get_cursor**(const struct *idr* *idr)

Return the current position of the cyclic allocator

Parameters

const struct idr *idr
idr handle

Description

The value returned is the value that will be next returned from idr_alloc_cyclic() if it is free (otherwise the search will start from this position).

void **idr_set_cursor**(struct *idr* *idr, unsigned int val)

Set the current position of the cyclic allocator

Parameters

struct idr *idr
idr handle

unsigned int val
new position

Description

The next call to idr_alloc_cyclic() will return **val** if it is free (otherwise the search will start from this position).

void **idr_init_base**(struct *idr* *idr, int base)

Initialise an IDR.

Parameters

struct idr *idr
IDR handle.

int base
The base value for the IDR.

Description

This variation of idr_init() creates an IDR which will allocate IDs starting at `base`.

void **idr_init**(struct *idr* *idr)

Initialise an IDR.

Parameters

struct idr *idr
IDR handle.

Description

Initialise a dynamically allocated IDR. To initialise a statically allocated IDR, use DEFINE_IDR().

bool **idr_is_empty**(const struct *idr* *idr)

Are there any IDs allocated?

Parameters

const struct idr *idr
IDR handle.

Return

`true` if any IDs have been allocated from this IDR.

void **idr_preload_end**(void)

end preload section started with idr_preload()

Parameters

void
no arguments

Description

Each idr_preload() should be matched with an invocation of this function. See idr_preload() for details.

idr_for_each_entry

`idr_for_each_entry (idr, entry, id)`

Iterate over an IDR's elements of a given type.

Parameters

idr
IDR handle.

entry
The type * to use as cursor

id
Entry ID.

Description

entry and **id** do not need to be initialized before the loop, and after normal termination **entry** is left with the value NULL. This is convenient for a "not found" value.

idr_for_each_entry_ul

`idr_for_each_entry_ul (idr, entry, tmp, id)`

Iterate over an IDR's elements of a given type.

Parameters

idr
IDR handle.

entry
The type * to use as cursor.

tmp
A temporary placeholder for ID.

id
Entry ID.

Description

entry and **id** do not need to be initialized before the loop, and after normal termination **entry** is left with the value NULL. This is convenient for a "not found" value.

idr_for_each_entry_continue

`idr_for_each_entry_continue (idr, entry, id)`

Continue iteration over an IDR's elements of a given type

Parameters

idr
IDR handle.

entry
The type * to use as a cursor.

id
Entry ID.

Description

Continue to iterate over entries, continuing after the current position.

idr_for_each_entry_continue_ul

`idr_for_each_entry_continue_ul (idr, entry, tmp, id)`

Continue iteration over an IDR's elements of a given type

Parameters

idr
IDR handle.

entry
The type * to use as a cursor.

tmp
A temporary placeholder for ID.

id
Entry ID.

Description

Continue to iterate over entries, continuing after the current position. After normal termination **entry** is left with the value NULL. This is convenient for a "not found" value.

int **ida_alloc**(struct *ida* *ida, gfp_t gfp)
Allocate an unused ID.

Parameters

struct ida *ida
IDA handle.

gfp_t gfp
Memory allocation flags.

Description

Allocate an ID between 0 and `INT_MAX`, inclusive.

Context

Any context. It is safe to call this function without locking in your code.

Return

The allocated ID, or `-ENOMEM` if memory could not be allocated, or `-ENOSPC` if there are no free IDs.

int **ida_alloc_min**(struct *ida* *ida, unsigned int min, gfp_t gfp)
Allocate an unused ID.

Parameters

struct ida *ida
IDA handle.

unsigned int min
Lowest ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Allocate an ID between **min** and `INT_MAX`, inclusive.

Context

Any context. It is safe to call this function without locking in your code.

Return

The allocated ID, or -ENOMEM if memory could not be allocated, or -ENOSPC if there are no free IDs.

int **ida_alloc_max**(struct *ida* *ida, unsigned int max, gfp_t gfp)

Allocate an unused ID.

Parameters

struct ida *ida
IDA handle.

unsigned int max
Highest ID to allocate.

gfp_t gfp
Memory allocation flags.

Description

Allocate an ID between 0 and **max**, inclusive.

Context

Any context. It is safe to call this function without locking in your code.

Return

The allocated ID, or -ENOMEM if memory could not be allocated, or -ENOSPC if there are no free IDs.

int **idr_alloc_u32**(struct *idr* *idr, void *ptr, u32 *nextid, unsigned long max, gfp_t gfp)

Allocate an ID.

Parameters

struct idr *idr
IDR handle.

void *ptr
Pointer to be associated with the new ID.

u32 *nextid
Pointer to an ID.

unsigned long max
The maximum ID to allocate (inclusive).

gfp_t gfp
Memory allocation flags.

Description

Allocates an unused ID in the range specified by **nextid** and **max**. Note that **max** is inclusive whereas the **end** parameter to idr_alloc() is exclusive. The new ID is assigned to **nextid** before the pointer is inserted into the IDR, so if **nextid** points into the object pointed to by **ptr**, a concurrent lookup will not find an uninitialised ID.

The caller should provide their own locking to ensure that two concurrent modifications to the IDR are not possible. Read-only accesses to the IDR may be done under the RCU read lock or may exclude simultaneous writers.

Return

0 if an ID was allocated, -ENOMEM if memory allocation failed, or -ENOSPC if no free IDs could be found. If an error occurred, **nextid** is unchanged.

int **idr_alloc**(struct *idr* *idr, void *ptr, int start, int end, gfp_t gfp)

Allocate an ID.

Parameters

struct idr *idr
IDR handle.

void *ptr
Pointer to be associated with the new ID.

int start
The minimum ID (inclusive).

int end
The maximum ID (exclusive).

gfp_t gfp
Memory allocation flags.

Description

Allocates an unused ID in the range specified by **start** and **end**. If **end** is <= 0, it is treated as one larger than `INT_MAX`. This allows callers to use **start** + N as **end** as long as N is within integer range.

The caller should provide their own locking to ensure that two concurrent modifications to the IDR are not possible. Read-only accesses to the IDR may be done under the RCU read lock or may exclude simultaneous writers.

Return

The newly allocated ID, -ENOMEM if memory allocation failed, or -ENOSPC if no free IDs could be found.

int **idr_alloc_cyclic**(struct *idr* *idr, void *ptr, int start, int end, gfp_t gfp)

Allocate an ID cyclically.

Parameters

struct idr *idr
IDR handle.

void *ptr
Pointer to be associated with the new ID.

int start
The minimum ID (inclusive).

int end
The maximum ID (exclusive).

gfp_t gfp
Memory allocation flags.

Description

Allocates an unused ID in the range specified by **start** and **end**. If **end** is <= 0, it is treated as one larger than `INT_MAX`. This allows callers to use **start** + N as **end** as long as N is within integer range. The search for an unused ID will start at the last ID allocated and will wrap around to **start** if no free IDs are found before reaching **end**.

The caller should provide their own locking to ensure that two concurrent modifications to the IDR are not possible. Read-only accesses to the IDR may be done under the RCU read lock or may exclude simultaneous writers.

Return

The newly allocated ID, -ENOMEM if memory allocation failed, or -ENOSPC if no free IDs could be found.

void ***idr_remove**(struct *idr* *idr, unsigned long id)

Remove an ID from the IDR.

Parameters

struct idr *idr
IDR handle.

unsigned long id
Pointer ID.

Description

Removes this ID from the IDR. If the ID was not previously in the IDR, this function returns `NULL`.

Since this function modifies the IDR, the caller should provide their own locking to ensure that concurrent modification of the same IDR is not possible.

Return

The pointer formerly associated with this ID.

void ***idr_find**(const struct *idr* *idr, unsigned long id)

Return pointer for given ID.

Parameters

const struct idr *idr
IDR handle.

unsigned long id
Pointer ID.

Description

Looks up the pointer associated with this ID. A `NULL` pointer may indicate that **id** is not allocated or that the `NULL` pointer was associated with this ID.

This function can be called under rcu_read_lock(), given that the leaf pointers lifetimes are correctly managed.

Return

The pointer associated with this ID.

int **idr_for_each**(const struct *idr* *idr, int (*fn)(int id, void *p, void *data), void *data)

Iterate through all stored pointers.

Parameters

const struct idr *idr
IDR handle.

int (*fn)(int id, void *p, void *data)
Function to be called for each pointer.

void *data
Data passed to callback function.

Description

The callback function will be called for each entry in **idr**, passing the ID, the entry and **data**.

If **fn** returns anything other than `0`, the iteration stops and that value is returned from this function.

idr_for_each() can be called concurrently with idr_alloc() and idr_remove() if protected by RCU. Newly added entries may not be seen and deleted entries may be seen, but adding and removing entries will not cause other entries to be skipped, nor spurious ones to be seen.

void ***idr_get_next_ul**(struct *idr* *idr, unsigned long *nextid)

Find next populated entry.

Parameters

struct idr *idr
IDR handle.

unsigned long *nextid
Pointer to an ID.

Description

Returns the next populated entry in the tree with an ID greater than or equal to the value pointed to by **nextid**. On exit, **nextid** is updated to the ID of the found value. To use in a loop, the value pointed to by nextid must be incremented by the user.

void ***idr_get_next**(struct *idr* *idr, int *nextid)

Find next populated entry.

Parameters

struct idr *idr
IDR handle.

int *nextid
Pointer to an ID.

Description

Returns the next populated entry in the tree with an ID greater than or equal to the value pointed to by **nextid**. On exit, **nextid** is updated to the ID of the found value. To use in a loop, the value pointed to by nextid must be incremented by the user.

void *`idr_replace`(struct *idr* *idr, void *ptr, unsigned long id)

replace pointer for given ID.

Parameters

`struct idr *idr`
IDR handle.

`void *ptr`
New pointer to associate with the ID.

`unsigned long id`
ID to change.

Description

Replace the pointer registered with an ID and return the old value. This function can be called under the RCU read lock concurrently with idr_alloc() and idr_remove() (as long as the ID being removed is not the one being replaced!).

Return

the old value on success. `-ENOENT` indicates that **id** was not found. `-EINVAL` indicates that **ptr** was not valid.

int `ida_alloc_range`(struct *ida* *ida, unsigned int min, unsigned int max, gfp_t gfp)

Allocate an unused ID.

Parameters

`struct ida *ida`
IDA handle.

`unsigned int min`
Lowest ID to allocate.

`unsigned int max`
Highest ID to allocate.

`gfp_t gfp`
Memory allocation flags.

Description

Allocate an ID between **min** and **max**, inclusive. The allocated ID will not exceed `INT_MAX`, even if **max** is larger.

Context

Any context. It is safe to call this function without locking in your code.

Return

The allocated ID, or `-ENOMEM` if memory could not be allocated, or `-ENOSPC` if there are no free IDs.

void `ida_free`(struct *ida* *ida, unsigned int id)

Release an allocated ID.

Parameters

struct ida *ida
IDA handle.

unsigned int id
Previously allocated ID.

Context

Any context. It is safe to call this function without locking in your code.

void **ida_destroy**(struct *ida* *ida)
Free all IDs.

Parameters

struct ida *ida
IDA handle.

Description

Calling this function frees all IDs and releases all resources used by an IDA. When this call returns, the IDA is empty and can be reused or freed. If the IDA is already empty, there is no need to call this function.

Context

Any context. It is safe to call this function without locking in your code.

2.7 Circular Buffers

Author
David Howells <dhowells@redhat.com>

Author
Paul E. McKenney <paulmck@linux.ibm.com>

Linux provides a number of features that can be used to implement circular buffering. There are two sets of such features:

(1) Convenience functions for determining information about power-of-2 sized buffers.

(2) Memory barriers for when the producer and the consumer of objects in the buffer don't want to share a lock.

To use these facilities, as discussed below, there needs to be just one producer and just one consumer. It is possible to handle multiple producers by serialising them, and to handle multiple consumers by serialising them.

2.7.1 What is a circular buffer?

First of all, what is a circular buffer? A circular buffer is a buffer of fixed, finite size into which there are two indices:

(1) A 'head' index - the point at which the producer inserts items into the buffer.

(2) A 'tail' index - the point at which the consumer finds the next item in the buffer.

Typically when the tail pointer is equal to the head pointer, the buffer is empty; and the buffer is full when the head pointer is one less than the tail pointer.

The head index is incremented when items are added, and the tail index when items are removed. The tail index should never jump the head index, and both indices should be wrapped to 0 when they reach the end of the buffer, thus allowing an infinite amount of data to flow through the buffer.

Typically, items will all be of the same unit size, but this isn't strictly required to use the techniques below. The indices can be increased by more than 1 if multiple items or variable-sized items are to be included in the buffer, provided that neither index overtakes the other. The implementer must be careful, however, as a region more than one unit in size may wrap the end of the buffer and be broken into two segments.

2.7.2 Measuring power-of-2 buffers

Calculation of the occupancy or the remaining capacity of an arbitrarily sized circular buffer would normally be a slow operation, requiring the use of a modulus (divide) instruction. However, if the buffer is of a power-of-2 size, then a much quicker bitwise-AND instruction can be used instead.

Linux provides a set of macros for handling power-of-2 circular buffers. These can be made use of by:

```
#include <linux/circ_buf.h>
```

The macros are:

(1) Measure the remaining capacity of a buffer:

```
CIRC_SPACE(head_index, tail_index, buffer_size);
```

This returns the amount of space left in the buffer[1] into which items can be inserted.

(2) Measure the maximum consecutive immediate space in a buffer:

```
CIRC_SPACE_TO_END(head_index, tail_index, buffer_size);
```

This returns the amount of consecutive space left in the buffer[1] into which items can be immediately inserted without having to wrap back to the beginning of the buffer.

(3) Measure the occupancy of a buffer:

```
CIRC_CNT(head_index, tail_index, buffer_size);
```

This returns the number of items currently occupying a buffer[2].

(4) Measure the non-wrapping occupancy of a buffer:

```
CIRC_CNT_TO_END(head_index, tail_index, buffer_size);
```

This returns the number of consecutive items[2] that can be extracted from the buffer without having to wrap back to the beginning of the buffer.

Each of these macros will nominally return a value between 0 and buffer_size-1, however:

(1) CIRC_SPACE*() are intended to be used in the producer. To the producer they will return a lower bound as the producer controls the head index, but the consumer may still be depleting the buffer on another CPU and moving the tail index.

To the consumer it will show an upper bound as the producer may be busy depleting the space.

(2) CIRC_CNT*() are intended to be used in the consumer. To the consumer they will return a lower bound as the consumer controls the tail index, but the producer may still be filling the buffer on another CPU and moving the head index.

To the producer it will show an upper bound as the consumer may be busy emptying the buffer.

(3) To a third party, the order in which the writes to the indices by the producer and consumer become visible cannot be guaranteed as they are independent and may be made on different CPUs - so the result in such a situation will merely be a guess, and may even be negative.

2.7.3 Using memory barriers with circular buffers

By using memory barriers in conjunction with circular buffers, you can avoid the need to:

(1) use a single lock to govern access to both ends of the buffer, thus allowing the buffer to be filled and emptied at the same time; and

(2) use atomic counter operations.

There are two sides to this: the producer that fills the buffer, and the consumer that empties it. Only one thing should be filling a buffer at any one time, and only one thing should be emptying a buffer at any one time, but the two sides can operate simultaneously.

The producer

The producer will look something like this:

```
spin_lock(&producer_lock);

unsigned long head = buffer->head;
/* The spin_unlock() and next spin_lock() provide needed ordering. */
unsigned long tail = READ_ONCE(buffer->tail);

if (CIRC_SPACE(head, tail, buffer->size) >= 1) {
        /* insert one item into the buffer */
        struct item *item = buffer[head];

        produce_item(item);
```

```
        smp_store_release(buffer->head,
                          (head + 1) & (buffer->size - 1));

        /* wake_up() will make sure that the head is committed before
         * waking anyone up */
        wake_up(consumer);
}

spin_unlock(&producer_lock);
```

This will instruct the CPU that the contents of the new item must be written before the head index makes it available to the consumer and then instructs the CPU that the revised head index must be written before the consumer is woken.

Note that wake_up() does not guarantee any sort of barrier unless something is actually awakened. We therefore cannot rely on it for ordering. However, there is always one element of the array left empty. Therefore, the producer must produce two elements before it could possibly corrupt the element currently being read by the consumer. Therefore, the unlock-lock pair between consecutive invocations of the consumer provides the necessary ordering between the read of the index indicating that the consumer has vacated a given element and the write by the producer to that same element.

The Consumer

The consumer will look something like this:

```
spin_lock(&consumer_lock);

/* Read index before reading contents at that index. */
unsigned long head = smp_load_acquire(buffer->head);
unsigned long tail = buffer->tail;

if (CIRC_CNT(head, tail, buffer->size) >= 1) {

        /* extract one item from the buffer */
        struct item *item = buffer[tail];

        consume_item(item);

        /* Finish reading descriptor before incrementing tail. */
        smp_store_release(buffer->tail,
                          (tail + 1) & (buffer->size - 1));
}

spin_unlock(&consumer_lock);
```

This will instruct the CPU to make sure the index is up to date before reading the new item, and then it shall make sure the CPU has finished reading the item before it writes the new tail pointer, which will erase the item.

Note the use of READ_ONCE() and smp_load_acquire() to read the opposition index. This prevents the compiler from discarding and reloading its cached value. This isn't strictly needed if you can be sure that the opposition index will _only_ be used the once. The smp_load_acquire() additionally forces the CPU to order against subsequent memory references. Similarly, smp_store_release() is used in both algorithms to write the thread's index. This documents the fact that we are writing to something that can be read concurrently, prevents the compiler from tearing the store, and enforces ordering against previous accesses.

2.7.4 Further reading

See also Documentation/memory-barriers.txt for a description of Linux's memory barrier facilities.

2.8 Red-black Trees (rbtree) in Linux

Date
January 18, 2007

Author
Rob Landley <rob@landley.net>

2.8.1 What are red-black trees, and what are they for?

Red-black trees are a type of self-balancing binary search tree, used for storing sortable key/value data pairs. This differs from radix trees (which are used to efficiently store sparse arrays and thus use long integer indexes to insert/access/delete nodes) and hash tables (which are not kept sorted to be easily traversed in order, and must be tuned for a specific size and hash function where rbtrees scale gracefully storing arbitrary keys).

Red-black trees are similar to AVL trees, but provide faster real-time bounded worst case performance for insertion and deletion (at most two rotations and three rotations, respectively, to balance the tree), with slightly slower (but still O(log n)) lookup time.

To quote Linux Weekly News:

> There are a number of red-black trees in use in the kernel. The deadline and CFQ I/O schedulers employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the "hierarchical token bucket" scheduler.

This document covers use of the Linux rbtree implementation. For more information on the nature and implementation of Red Black Trees, see:

Linux Weekly News article on red-black trees
https://lwn.net/Articles/184495/

Wikipedia entry on red-black trees
https://en.wikipedia.org/wiki/Red-black_tree

2.8.2 Linux implementation of red-black trees

Linux's rbtree implementation lives in the file "lib/rbtree.c". To use it, "#include <linux/rbtree.h>".

The Linux rbtree implementation is optimized for speed, and thus has one less layer of indirection (and better cache locality) than more traditional tree implementations. Instead of using pointers to separate rb_node and data structures, each instance of struct rb_node is embedded in the data structure it organizes. And instead of using a comparison callback function pointer, users are expected to write their own tree search and insert functions which call the provided rbtree functions. Locking is also left up to the user of the rbtree code.

2.8.3 Creating a new rbtree

Data nodes in an rbtree tree are structures containing a struct rb_node member:

```
struct mytype {
      struct rb_node node;
      char *keystring;
};
```

When dealing with a pointer to the embedded struct rb_node, the containing data structure may be accessed with the standard container_of() macro. In addition, individual members may be accessed directly via rb_entry(node, type, member).

At the root of each rbtree is an rb_root structure, which is initialized to be empty via:

> struct rb_root mytree = RB_ROOT;

2.8.4 Searching for a value in an rbtree

Writing a search function for your tree is fairly straightforward: start at the root, compare each value, and follow the left or right branch as necessary.

Example:

```
struct mytype *my_search(struct rb_root *root, char *string)
{
      struct rb_node *node = root->rb_node;

      while (node) {
              struct mytype *data = container_of(node, struct mytype, node);
              int result;

              result = strcmp(string, data->keystring);

              if (result < 0)
                      node = node->rb_left;
              else if (result > 0)
                      node = node->rb_right;
              else
                      return data;
```

```
	}
	return NULL;
}
```

2.8.5 Inserting data into an rbtree

Inserting data in the tree involves first searching for the place to insert the new node, then inserting the node and rebalancing ("recoloring") the tree.

The search for insertion differs from the previous search by finding the location of the pointer on which to graft the new node. The new node also needs a link to its parent node for rebalancing purposes.

Example:

```
int my_insert(struct rb_root *root, struct mytype *data)
{
	struct rb_node **new = &(root->rb_node), *parent = NULL;

	/* Figure out where to put new node */
	while (*new) {
		struct mytype *this = container_of(*new, struct mytype, node);
		int result = strcmp(data->keystring, this->keystring);

		parent = *new;
		if (result < 0)
			new = &((*new)->rb_left);
		else if (result > 0)
			new = &((*new)->rb_right);
		else
			return FALSE;
	}

	/* Add new node and rebalance tree. */
	rb_link_node(&data->node, parent, new);
	rb_insert_color(&data->node, root);

	return TRUE;
}
```

2.8.6 Removing or replacing existing data in an rbtree

To remove an existing node from a tree, call:

```
void rb_erase(struct rb_node *victim, struct rb_root *tree);
```

Example:

```
struct mytype *data = mysearch(&mytree, "walrus");
```

```
if (data) {
      rb_erase(&data->node, &mytree);
      myfree(data);
}
```

To replace an existing node in a tree with a new one with the same key, call:

```
void rb_replace_node(struct rb_node *old, struct rb_node *new,
                     struct rb_root *tree);
```

Replacing a node this way does not re-sort the tree: If the new node doesn't have the same key as the old node, the rbtree will probably become corrupted.

2.8.7 Iterating through the elements stored in an rbtree (in sort order)

Four functions are provided for iterating through an rbtree's contents in sorted order. These work on arbitrary trees, and should not need to be modified or wrapped (except for locking purposes):

```
struct rb_node *rb_first(struct rb_root *tree);
struct rb_node *rb_last(struct rb_root *tree);
struct rb_node *rb_next(struct rb_node *node);
struct rb_node *rb_prev(struct rb_node *node);
```

To start iterating, call rb_first() or rb_last() with a pointer to the root of the tree, which will return a pointer to the node structure contained in the first or last element in the tree. To continue, fetch the next or previous node by calling rb_next() or rb_prev() on the current node. This will return NULL when there are no more nodes left.

The iterator functions return a pointer to the embedded struct rb_node, from which the containing data structure may be accessed with the container_of() macro, and individual members may be accessed directly via rb_entry(node, type, member).

Example:

```
struct rb_node *node;
for (node = rb_first(&mytree); node; node = rb_next(node))
      printk("key=%s\n", rb_entry(node, struct mytype, node)->keystring);
```

2.8.8 Cached rbtrees

Computing the leftmost (smallest) node is quite a common task for binary search trees, such as for traversals or users relying on a the particular order for their own logic. To this end, users can use 'struct rb_root_cached' to optimize O(logN) rb_first() calls to a simple pointer fetch avoiding potentially expensive tree iterations. This is done at negligible runtime overhead for maintenance; albeit larger memory footprint.

Similar to the rb_root structure, cached rbtrees are initialized to be empty via:

```
struct rb_root_cached mytree = RB_ROOT_CACHED;
```

Cached rbtree is simply a regular rb_root with an extra pointer to cache the leftmost node. This allows rb_root_cached to exist wherever rb_root does, which permits augmented trees to be supported as well as only a few extra interfaces:

```
struct rb_node *rb_first_cached(struct rb_root_cached *tree);
void rb_insert_color_cached(struct rb_node *, struct rb_root_cached *, bool);
void rb_erase_cached(struct rb_node *node, struct rb_root_cached *);
```

Both insert and erase calls have their respective counterpart of augmented trees:

```
void rb_insert_augmented_cached(struct rb_node *node, struct rb_root_cached *,
                                bool, struct rb_augment_callbacks *);
void rb_erase_augmented_cached(struct rb_node *, struct rb_root_cached *,
                               struct rb_augment_callbacks *);
```

2.8.9 Support for Augmented rbtrees

Augmented rbtree is an rbtree with "some" additional data stored in each node, where the additional data for node N must be a function of the contents of all nodes in the subtree rooted at N. This data can be used to augment some new functionality to rbtree. Augmented rbtree is an optional feature built on top of basic rbtree infrastructure. An rbtree user who wants this feature will have to call the augmentation functions with the user provided augmentation callback when inserting and erasing nodes.

C files implementing augmented rbtree manipulation must include <linux/rbtree_augmented.h> instead of <linux/rbtree.h>. Note that linux/rbtree_augmented.h exposes some rbtree implementations details you are not expected to rely on; please stick to the documented APIs there and do not include <linux/rbtree_augmented.h> from header files either so as to minimize chances of your users accidentally relying on such implementation details.

On insertion, the user must update the augmented information on the path leading to the inserted node, then call rb_link_node() as usual and rb_augment_inserted() instead of the usual rb_insert_color() call. If rb_augment_inserted() rebalances the rbtree, it will callback into a user provided function to update the augmented information on the affected subtrees.

When erasing a node, the user must call rb_erase_augmented() instead of rb_erase(). rb_erase_augmented() calls back into user provided functions to updated the augmented information on affected subtrees.

In both cases, the callbacks are provided through struct rb_augment_callbacks. 3 callbacks must be defined:

- A propagation callback, which updates the augmented value for a given node and its ancestors, up to a given stop point (or NULL to update all the way to the root).
- A copy callback, which copies the augmented value for a given subtree to a newly assigned subtree root.
- A tree rotation callback, which copies the augmented value for a given subtree to a newly assigned subtree root AND recomputes the augmented information for the former subtree root.

The compiled code for rb_erase_augmented() may inline the propagation and copy callbacks, which results in a large function, so each augmented rbtree user should have a single rb_erase_augmented() call site in order to limit compiled code size.

Sample usage

Interval tree is an example of augmented rb tree. Reference - "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein. More details about interval trees:

Classical rbtree has a single key and it cannot be directly used to store interval ranges like [lo:hi] and do a quick lookup for any overlap with a new lo:hi or to find whether there is an exact match for a new lo:hi.

However, rbtree can be augmented to store such interval ranges in a structured way making it possible to do efficient lookup and exact match.

This "extra information" stored in each node is the maximum hi (max_hi) value among all the nodes that are its descendants. This information can be maintained at each node just be looking at the node and its immediate children. And this will be used in O(log n) lookup for lowest match (lowest start address among all possible matches) with something like:

```
struct interval_tree_node *
interval_tree_first_match(struct rb_root *root,
                          unsigned long start, unsigned long last)
{
        struct interval_tree_node *node;

        if (!root->rb_node)
                return NULL;
        node = rb_entry(root->rb_node, struct interval_tree_node, rb);

        while (true) {
                if (node->rb.rb_left) {
                        struct interval_tree_node *left =
                                rb_entry(node->rb.rb_left,
                                         struct interval_tree_node, rb);
                        if (left->__subtree_last >= start) {
                                /*
                                 * Some nodes in left subtree satisfy Cond2.
                                 * Iterate to find the leftmost such node N.
                                 * If it also satisfies Cond1, that's the match
                                 * we are looking for. Otherwise, there is no
                                 * matching interval as nodes to the right of N
                                 * can't satisfy Cond1 either.
                                 */
                                node = left;
                                continue;
                        }
                }
                if (node->start <= last) {              /* Cond1 */
                        if (node->last >= start)        /* Cond2 */
                                return node;    /* node is leftmost match */
```

```
			if (node->rb.rb_right) {
				node = rb_entry(node->rb.rb_right,
					struct interval_tree_node, rb);
				if (node->__subtree_last >= start)
					continue;
			}
		}
		return NULL;	/* No match */
	}
}
```

Insertion/removal are defined using the following augmented callbacks:

```
static inline unsigned long
compute_subtree_last(struct interval_tree_node *node)
{
	unsigned long max = node->last, subtree_last;
	if (node->rb.rb_left) {
		subtree_last = rb_entry(node->rb.rb_left,
			struct interval_tree_node, rb)->__subtree_last;
		if (max < subtree_last)
			max = subtree_last;
	}
	if (node->rb.rb_right) {
		subtree_last = rb_entry(node->rb.rb_right,
			struct interval_tree_node, rb)->__subtree_last;
		if (max < subtree_last)
			max = subtree_last;
	}
	return max;
}

static void augment_propagate(struct rb_node *rb, struct rb_node *stop)
{
	while (rb != stop) {
		struct interval_tree_node *node =
			rb_entry(rb, struct interval_tree_node, rb);
		unsigned long subtree_last = compute_subtree_last(node);
		if (node->__subtree_last == subtree_last)
			break;
		node->__subtree_last = subtree_last;
		rb = rb_parent(&node->rb);
	}
}

static void augment_copy(struct rb_node *rb_old, struct rb_node *rb_new)
{
	struct interval_tree_node *old =
		rb_entry(rb_old, struct interval_tree_node, rb);
	struct interval_tree_node *new =
		rb_entry(rb_new, struct interval_tree_node, rb);
```

```
	new->__subtree_last = old->__subtree_last;
}

static void augment_rotate(struct rb_node *rb_old, struct rb_node *rb_new)
{
	struct interval_tree_node *old =
		rb_entry(rb_old, struct interval_tree_node, rb);
	struct interval_tree_node *new =
		rb_entry(rb_new, struct interval_tree_node, rb);

	new->__subtree_last = old->__subtree_last;
	old->__subtree_last = compute_subtree_last(old);
}

static const struct rb_augment_callbacks augment_callbacks = {
	augment_propagate, augment_copy, augment_rotate
};

void interval_tree_insert(struct interval_tree_node *node,
			  struct rb_root *root)
{
	struct rb_node **link = &root->rb_node, *rb_parent = NULL;
	unsigned long start = node->start, last = node->last;
	struct interval_tree_node *parent;

	while (*link) {
		rb_parent = *link;
		parent = rb_entry(rb_parent, struct interval_tree_node, rb);
		if (parent->__subtree_last < last)
			parent->__subtree_last = last;
		if (start < parent->start)
			link = &parent->rb.rb_left;
		else
			link = &parent->rb.rb_right;
	}

	node->__subtree_last = last;
	rb_link_node(&node->rb, rb_parent, link);
	rb_insert_augmented(&node->rb, root, &augment_callbacks);
}

void interval_tree_remove(struct interval_tree_node *node,
			  struct rb_root *root)
{
	rb_erase_augmented(&node->rb, root, &augment_callbacks);
}
```

2.9 Generic radix trees/sparse arrays

Very simple and minimalistic, supporting arbitrary size entries up to PAGE_SIZE.

A genradix is defined with the type it will store, like so:

static GENRADIX(struct foo) foo_genradix;

The main operations are:

- genradix_init(radix) - initialize an empty genradix
- genradix_free(radix) - free all memory owned by the genradix and reinitialize it
- genradix_ptr(radix, idx) - gets a pointer to the entry at idx, returning NULL if that entry does not exist
- genradix_ptr_alloc(radix, idx, gfp) - gets a pointer to an entry, allocating it if necessary
- genradix_for_each(radix, iter, p) - iterate over each entry in a genradix

The radix tree allocates one page of entries at a time, so entries may exist that were never explicitly allocated - they will be initialized to all zeroes.

Internally, a genradix is just a radix tree of pages, and indexing works in terms of byte offsets. The wrappers in this header file use sizeof on the type the radix contains to calculate a byte offset from the index - see __idx_to_offset.

2.9.1 generic radix tree functions

genradix_init

`genradix_init (_radix)`

initialize a genradix

Parameters

_radix
genradix to initialize

Description

Does not fail

genradix_free

`genradix_free (_radix)`

free all memory owned by a genradix

Parameters

_radix
the genradix to free

Description

After freeing, **_radix** will be reinitialized and empty

genradix_ptr

`genradix_ptr (_radix, _idx)`

get a pointer to a genradix entry

Parameters

_radix
genradix to access

_idx
index to fetch

Description

Returns a pointer to entry at **_idx**, or NULL if that entry does not exist.

genradix_ptr_alloc

`genradix_ptr_alloc (_radix, _idx, _gfp)`

get a pointer to a genradix entry, allocating it if necessary

Parameters

_radix
genradix to access

_idx
index to fetch

_gfp
gfp mask

Description

Returns a pointer to entry at **_idx**, or NULL on allocation failure

genradix_iter_init

`genradix_iter_init (_radix, _idx)`

initialize a genradix_iter

Parameters

_radix
genradix that will be iterated over

_idx
index to start iterating from

genradix_iter_peek

`genradix_iter_peek (_iter, _radix)`

get first entry at or above iterator's current position

Parameters

_iter
a genradix_iter

_radix
genradix being iterated over

Description

If no more entries exist at or above **_iter**'s current position, returns NULL

genradix_for_each

`genradix_for_each (_radix, _iter, _p)`

iterate over entry in a genradix

Parameters

_radix
genradix to iterate over

_iter
a genradix_iter to track current position

_p
pointer to genradix entry type

Description

On every iteration, **_p** will point to the current entry, and **_iter.pos** will be the current entry's index.

genradix_prealloc

`genradix_prealloc (_radix, _nr, _gfp)`

preallocate entries in a generic radix tree

Parameters

_radix
genradix to preallocate

_nr
number of entries to preallocate

_gfp
gfp mask

Description

Returns 0 on success, -ENOMEM on failure

2.10 Generic bitfield packing and unpacking functions

2.10.1 Problem statement

When working with hardware, one has to choose between several approaches of interfacing with it. One can memory-map a pointer to a carefully crafted struct over the hardware device's memory region, and access its fields as struct members (potentially declared as bitfields). But writing code this way would make it less portable, due to potential endianness mismatches between the CPU and the hardware device. Additionally, one has to pay close attention when translating register definitions from the hardware documentation into bit field indices for the structs. Also, some hardware (typically networking equipment) tends to group its register fields in ways that violate any reasonable word boundaries (sometimes even 64 bit ones). This creates the inconvenience of having to define "high" and "low" portions of register fields within the struct. A more robust alternative to struct field definitions would be to extract the required fields by shifting the appropriate number of bits. But this would still not protect from endianness mismatches, except if all memory accesses were performed byte-by-byte. Also the code can easily get cluttered, and the high-level idea might get lost among the many bit shifts required. Many drivers take the bit-shifting approach and then attempt to reduce the clutter with tailored macros, but more often than not these macros take shortcuts that still prevent the code from being truly portable.

2.10.2 The solution

This API deals with 2 basic operations:

- Packing a CPU-usable number into a memory buffer (with hardware constraints/quirks)
- Unpacking a memory buffer (which has hardware constraints/quirks) into a CPU-usable number.

The API offers an abstraction over said hardware constraints and quirks, over CPU endianness and therefore between possible mismatches between the two.

The basic unit of these API functions is the u64. From the CPU's perspective, bit 63 always means bit offset 7 of byte 7, albeit only logically. The question is: where do we lay this bit out in memory?

The following examples cover the memory layout of a packed u64 field. The byte offsets in the packed buffer are always implicitly 0, 1, ... 7. What the examples show is where the logical bytes and bits sit.

1. Normally (no quirks), we would do it like this:

```
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38␣
↪37 36 35 34 33 32
7                       6                       5                       4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6 ␣
↪5  4  3  2  1  0
3                       2                       1                       0
```

That is, the MSByte (7) of the CPU-usable u64 sits at memory offset 0, and the LSByte (0) of the u64 sits at memory offset 7. This corresponds to what most folks would regard to as "big endian", where bit i corresponds to the number 2^i. This is also referred to in the code comments as "logical" notation.

2. If QUIRK_MSB_ON_THE_RIGHT is set, we do it like this:

```
56 57 58 59 60 61 62 63 48 49 50 51 52 53 54 55 40 41 42 43 44 45 46 47 32 33␣
↪34 35 36 37 38 39
7                       6                        5                      4
24 25 26 27 28 29 30 31 16 17 18 19 20 21 22 23  8  9 10 11 12 13 14 15  0  1 ␣
↪2  3  4  5  6  7
3                       2                        1                      0
```

That is, QUIRK_MSB_ON_THE_RIGHT does not affect byte positioning, but inverts bit offsets inside a byte.

3. If QUIRK_LITTLE_ENDIAN is set, we do it like this:

```
39 38 37 36 35 34 33 32 47 46 45 44 43 42 41 40 55 54 53 52 51 50 49 48 63 62 61 60 59 58 57 56
4                       5                       6                       7
7  6  5  4  3  2  1  0  15 14 13 12 11 10  9  8 23 22 21 20 19 18 17 16 31 30 29 28 27 26 25 24
0                       1                       2                       3
```

Therefore, QUIRK_LITTLE_ENDIAN means that inside the memory region, every byte from each 4-byte word is placed at its mirrored position compared to the boundary of that word.

4. If QUIRK_MSB_ON_THE_RIGHT and QUIRK_LITTLE_ENDIAN are both set, we do it like this:

```
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
4                       5                       6                       7
0  1  2  3  4  5  6  7  8   9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0                       1                       2                       3
```

5. If just QUIRK_LSW32_IS_FIRST is set, we do it like this:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
3                       2                       1                          0
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
7                       6                       5                          4
```

In this case the 8 byte memory region is interpreted as follows: first 4 bytes correspond to the least significant 4-byte word, next 4 bytes to the more significant 4-byte word.

6. If QUIRK_LSW32_IS_FIRST and QUIRK_MSB_ON_THE_RIGHT are set, we do it like this:

```
24 25 26 27 28 29 30 31 16 17 18 19 20 21 22 23  8  9 10 11 12 13 14 15  0  1  2  3  4  5  6  7
3                       2                        1                        0
56 57 58 59 60 61 62 63 48 49 50 51 52 53 54 55 40 41 42 43 44 45 46 47 32 33 34 35 36 37 38 39
7                       6                        5                        4
```

7. If QUIRK_LSW32_IS_FIRST and QUIRK_LITTLE_ENDIAN are set, it looks like this:

```
7  6  5  4  3  2  1  0  15 14 13 12 11 10  9  8 23 22 21 20 19 18 17 16 31 30 29 28 27 26 25 24
0                       1                       2                       3
39 38 37 36 35 34 33 32 47 46 45 44 43 42 41 40 55 54 53 52 51 50 49 48 63 62 61 60 59 58 57 56
4                       5                       6                       7
```

8. If QUIRK_LSW32_IS_FIRST, QUIRK_LITTLE_ENDIAN and QUIRK_MSB_ON_THE_RIGHT are set, it looks like this:

```
0  1  2  3  4  5  6  7  8   9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25␣
↪26 27 28 29 30 31
0                       1                       2                       3
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57␣
↪58 59 60 61 62 63
4                       5                       6                       7
```

We always think of our offsets as if there were no quirk, and we translate them afterwards, before accessing the memory region.

2.10.3 Intended use

Drivers that opt to use this API first need to identify which of the above 3 quirk combinations (for a total of 8) match what the hardware documentation describes. Then they should wrap the packing() function, creating a new xxx_packing() that calls it using the proper QUIRK_* one-hot bits set.

The packing() function returns an int-encoded error code, which protects the programmer against incorrect API use. The errors are not expected to occur during runtime, therefore it is reasonable for xxx_packing() to return void and simply swallow those errors. Optionally it can dump stack or print the error description.

2.11 this_cpu operations

Author
Christoph Lameter, August 4th, 2014

Author
Pranith Kumar, Aug 2nd, 2014

this_cpu operations are a way of optimizing access to per cpu variables associated with the *currently* executing processor. This is done through the use of segment registers (or a dedicated register where the cpu permanently stored the beginning of the per cpu area for a specific processor).

this_cpu operations add a per cpu variable offset to the processor specific per cpu base and encode that operation in the instruction operating on the per cpu variable.

This means that there are no atomicity issues between the calculation of the offset and the operation on the data. Therefore it is not necessary to disable preemption or interrupts to ensure that the processor is not changed between the calculation of the address and the operation on the data.

Read-modify-write operations are of particular interest. Frequently processors have special lower latency instructions that can operate without the typical synchronization overhead, but still provide some sort of relaxed atomicity guarantees. The x86, for example, can execute RMW (Read Modify Write) instructions like inc/dec/cmpxchg without the lock prefix and the associated latency penalty.

Access to the variable without the lock prefix is not synchronized but synchronization is not necessary since we are dealing with per cpu data specific to the currently executing processor. Only the current processor should be accessing that variable and therefore there are no concurrency issues with other processors in the system.

Please note that accesses by remote processors to a per cpu area are exceptional situations and may impact performance and/or correctness (remote write operations) of local RMW operations via this_cpu_*.

The main use of the this_cpu operations has been to optimize counter operations.

The following this_cpu() operations with implied preemption protection are defined. These operations can be used without worrying about preemption and interrupts:

```
this_cpu_read(pcp)
this_cpu_write(pcp, val)
this_cpu_add(pcp, val)
this_cpu_and(pcp, val)
this_cpu_or(pcp, val)
this_cpu_add_return(pcp, val)
this_cpu_xchg(pcp, nval)
this_cpu_cmpxchg(pcp, oval, nval)
this_cpu_sub(pcp, val)
this_cpu_inc(pcp)
this_cpu_dec(pcp)
this_cpu_sub_return(pcp, val)
this_cpu_inc_return(pcp)
this_cpu_dec_return(pcp)
```

2.11.1 Inner working of this_cpu operations

On x86 the fs: or the gs: segment registers contain the base of the per cpu area. It is then possible to simply use the segment override to relocate a per cpu relative address to the proper per cpu area for the processor. So the relocation to the per cpu base is encoded in the instruction via a segment register prefix.

For example:

```
DEFINE_PER_CPU(int, x);
int z;

z = this_cpu_read(x);
```

results in a single instruction:

```
mov ax, gs:[x]
```

instead of a sequence of calculation of the address and then a fetch from that address which occurs with the per cpu operations. Before this_cpu_ops such sequence also required preempt disable/enable to prevent the kernel from moving the thread to a different processor while the calculation is performed.

Consider the following this_cpu operation:

```
this_cpu_inc(x)
```

The above results in the following single instruction (no lock prefix!):

```
inc gs:[x]
```

instead of the following operations required if there is no segment register:

```
int *y;
int cpu;

cpu = get_cpu();
y = per_cpu_ptr(&x, cpu);
(*y)++;
put_cpu();
```

Note that these operations can only be used on per cpu data that is reserved for a specific processor. Without disabling preemption in the surrounding code this_cpu_inc() will only guarantee that one of the per cpu counters is correctly incremented. However, there is no guarantee that the OS will not move the process directly before or after the this_cpu instruction is executed. In general this means that the value of the individual counters for each processor are meaningless. The sum of all the per cpu counters is the only value that is of interest.

Per cpu variables are used for performance reasons. Bouncing cache lines can be avoided if multiple processors concurrently go through the same code paths. Since each processor has its own per cpu variables no concurrent cache line updates take place. The price that has to be paid for this optimization is the need to add up the per cpu counters when the value of a counter is needed.

2.11.2 Special operations

```
y = this_cpu_ptr(&x)
```

Takes the offset of a per cpu variable (&x !) and returns the address of the per cpu variable that belongs to the currently executing processor. this_cpu_ptr avoids multiple steps that the common get_cpu/put_cpu sequence requires. No processor number is available. Instead, the offset of the local per cpu area is simply added to the per cpu offset.

Note that this operation is usually used in a code segment when preemption has been disabled. The pointer is then used to access local per cpu data in a critical section. When preemption is re-enabled this pointer is usually no longer useful since it may no longer point to per cpu data of the current processor.

2.11.3 Per cpu variables and offsets

Per cpu variables have *offsets* to the beginning of the per cpu area. They do not have addresses although they look like that in the code. Offsets cannot be directly dereferenced. The offset must be added to a base pointer of a per cpu area of a processor in order to form a valid address.

Therefore the use of x or &x outside of the context of per cpu operations is invalid and will generally be treated like a NULL pointer dereference.

```
DEFINE_PER_CPU(int, x);
```

In the context of per cpu operations the above implies that x is a per cpu variable. Most this_cpu operations take a cpu variable.

```
int __percpu *p = &x;
```

&x and hence p is the *offset* of a per cpu variable. this_cpu_ptr() takes the offset of a per cpu variable which makes this look a bit strange.

2.11.4 Operations on a field of a per cpu structure

Let's say we have a percpu structure:

```
struct s {
        int n,m;
};

DEFINE_PER_CPU(struct s, p);
```

Operations on these fields are straightforward:

```
this_cpu_inc(p.m)

z = this_cpu_cmpxchg(p.m, 0, 1);
```

If we have an offset to struct s:

```
struct s __percpu *ps = &p;

this_cpu_dec(ps->m);

z = this_cpu_inc_return(ps->n);
```

The calculation of the pointer may require the use of this_cpu_ptr() if we do not make use of this_cpu ops later to manipulate fields:

```
struct s *pp;

pp = this_cpu_ptr(&p);

pp->m--;

z = pp->n++;
```

2.11.5 Variants of this_cpu ops

this_cpu ops are interrupt safe. Some architectures do not support these per cpu local operations. In that case the operation must be replaced by code that disables interrupts, then does the operations that are guaranteed to be atomic and then re-enable interrupts. Doing so is expensive. If there are other reasons why the scheduler cannot change the processor we are executing on then there is no reason to disable interrupts. For that purpose the following __this_cpu operations are provided.

These operations have no guarantee against concurrent interrupts or preemption. If a per cpu variable is not used in an interrupt context and the scheduler cannot preempt, then they are safe. If any interrupts still occur while an operation is in progress and if the interrupt too modifies the variable, then RMW actions can not be guaranteed to be safe:

```
__this_cpu_read(pcp)
__this_cpu_write(pcp, val)
__this_cpu_add(pcp, val)
__this_cpu_and(pcp, val)
__this_cpu_or(pcp, val)
__this_cpu_add_return(pcp, val)
__this_cpu_xchg(pcp, nval)
__this_cpu_cmpxchg(pcp, oval, nval)
__this_cpu_sub(pcp, val)
__this_cpu_inc(pcp)
__this_cpu_dec(pcp)
__this_cpu_sub_return(pcp, val)
__this_cpu_inc_return(pcp)
__this_cpu_dec_return(pcp)
```

Will increment x and will not fall-back to code that disables interrupts on platforms that cannot accomplish atomicity through address relocation and a Read-Modify-Write operation in the same instruction.

2.11.6 &this_cpu_ptr(pp)->n vs this_cpu_ptr(&pp->n)

The first operation takes the offset and forms an address and then adds the offset of the n field. This may result in two add instructions emitted by the compiler.

The second one first adds the two offsets and then does the relocation. IMHO the second form looks cleaner and has an easier time with (). The second form also is consistent with the way this_cpu_read() and friends are used.

2.11.7 Remote access to per cpu data

Per cpu data structures are designed to be used by one cpu exclusively. If you use the variables as intended, this_cpu_ops() are guaranteed to be "atomic" as no other CPU has access to these data structures.

There are special cases where you might need to access per cpu data structures remotely. It is usually safe to do a remote read access and that is frequently done to summarize counters. Remote write access something which could be problematic because this_cpu ops do not have lock semantics. A remote write may interfere with a this_cpu RMW operation.

Remote write accesses to percpu data structures are highly discouraged unless absolutely necessary. Please consider using an IPI to wake up the remote CPU and perform the update to its per cpu area.

To access per-cpu data structure remotely, typically the per_cpu_ptr() function is used:

```
DEFINE_PER_CPU(struct data, datap);

struct data *p = per_cpu_ptr(&datap, cpu);
```

This makes it explicit that we are getting ready to access a percpu area remotely.

You can also do the following to convert the datap offset to an address:

```
struct data *p = this_cpu_ptr(&datap);
```

but, passing of pointers calculated via this_cpu_ptr to other cpus is unusual and should be avoided.

Remote access are typically only for reading the status of another cpus per cpu data. Write accesses can cause unique problems due to the relaxed synchronization requirements for this_cpu operations.

One example that illustrates some concerns with write operations is the following scenario that occurs because two per cpu variables share a cache-line but the relaxed synchronization is applied to only one process updating the cache-line.

Consider the following example:

```
struct test {
        atomic_t a;
        int b;
};

DEFINE_PER_CPU(struct test, onecacheline);
```

There is some concern about what would happen if the field 'a' is updated remotely from one processor and the local processor would use this_cpu ops to update field b. Care should be taken that such simultaneous accesses to data within the same cache line are avoided. Also costly synchronization may be necessary. IPIs are generally recommended in such scenarios instead of a remote write to the per cpu area of another processor.

Even in cases where the remote writes are rare, please bear in mind that a remote write will evict the cache line from the processor that most likely will access it. If the processor wakes up and finds a missing local cache line of a per cpu area, its performance and hence the wake up times will be affected.

2.12 ktime accessors

Device drivers can read the current time using ktime_get() and the many related functions declared in linux/timekeeping.h. As a rule of thumb, using an accessor with a shorter name is preferred over one with a longer name if both are equally fit for a particular use case.

2.12.1 Basic ktime_t based interfaces

The recommended simplest form returns an opaque ktime_t, with variants that return time for different clock references:

ktime_t **ktime_get**(void)

CLOCK_MONOTONIC

Useful for reliable timestamps and measuring short time intervals accurately. Starts at system boot time but stops during suspend.

ktime_t **ktime_get_boottime**(void)

CLOCK_BOOTTIME

Like ktime_get(), but does not stop when suspended. This can be used e.g. for key expiration times that need to be synchronized with other machines across a suspend operation.

ktime_t **ktime_get_real**(void)

CLOCK_REALTIME

Returns the time in relative to the UNIX epoch starting in 1970 using the Coordinated Universal Time (UTC), same as gettimeofday() user space. This is used for all timestamps that need to persist across a reboot, like inode times, but should be avoided for internal uses, since it can jump backwards due to a leap second update, NTP adjustment settimeofday() operation from user space.

ktime_t **ktime_get_clocktai**(void)

CLOCK_TAI

Like *ktime_get_real()*, but uses the International Atomic Time (TAI) reference instead of UTC to avoid jumping on leap second updates. This is rarely useful in the kernel.

ktime_t **ktime_get_raw**(void)

CLOCK_MONOTONIC_RAW

Like ktime_get(), but runs at the same rate as the hardware clocksource without (NTP) adjustments for clock drift. This is also rarely needed in the kernel.

2.12.2 nanosecond, timespec64, and second output

For all of the above, there are variants that return the time in a different format depending on what is required by the user:

u64 **ktime_get_ns**(void)

u64 **ktime_get_boottime_ns**(void)

u64 **ktime_get_real_ns**(void)

u64 **ktime_get_clocktai_ns**(void)

u64 **ktime_get_raw_ns**(void)

Same as the plain ktime_get functions, but returning a u64 number of nanoseconds in the respective time reference, which may be more convenient for some callers.

void **ktime_get_ts64**(struct timespec64*)

void **ktime_get_boottime_ts64**(struct timespec64*)

void **ktime_get_real_ts64**(struct timespec64*)

void **ktime_get_clocktai_ts64**(struct timespec64*)

void **ktime_get_raw_ts64**(struct timespec64*)

Same above, but returns the time in a 'struct timespec64', split into seconds and nanoseconds. This can avoid an extra division when printing the time, or when passing it into an external interface that expects a 'timespec' or 'timeval' structure.

time64_t **ktime_get_seconds**(void)

time64_t **ktime_get_boottime_seconds**(void)

time64_t **ktime_get_real_seconds**(void)

time64_t **ktime_get_clocktai_seconds**(void)

time64_t **ktime_get_raw_seconds**(void)

Return a coarse-grained version of the time as a scalar time64_t. This avoids accessing the clock hardware and rounds down the seconds to the full seconds of the last timer tick using the respective reference.

2.12.3 Coarse and fast_ns access

Some additional variants exist for more specialized cases:

ktime_t **ktime_get_coarse**(void)

ktime_t **ktime_get_coarse_boottime**(void)

ktime_t **ktime_get_coarse_real**(void)

ktime_t **ktime_get_coarse_clocktai**(void)

u64 **ktime_get_coarse_ns**(void)

u64 **ktime_get_coarse_boottime_ns**(void)

u64 **ktime_get_coarse_real_ns**(void)

u64 **ktime_get_coarse_clocktai_ns**(void)

void **ktime_get_coarse_ts64**(struct timespec64*)

void **ktime_get_coarse_boottime_ts64**(struct timespec64*)

void **ktime_get_coarse_real_ts64**(struct timespec64*) void **ktime_get_coarse_clocktai_ts64**(struct timespec64*)

These are quicker than the non-coarse versions, but less accurate, corresponding to CLOCK_MONOTONIC_COARSE and CLOCK_REALTIME_COARSE in user space, along with the equivalent boottime/tai/raw timebase not available in user space.

The time returned here corresponds to the last timer tick, which may be as much as 10ms in the past (for CONFIG_HZ=100), same as reading the 'jiffies' variable. These are only useful when called in a fast path and one still expects better than second accuracy, but can't easily use 'jiffies', e.g. for inode timestamps. Skipping the hardware clock access saves around 100 CPU cycles on most modern machines with a reliable cycle counter, but up to several microseconds on older hardware with an external clocksource.

u64 **ktime_get_mono_fast_ns**(void)

u64 **ktime_get_raw_fast_ns**(void)

u64 **ktime_get_boot_fast_ns**(void)

u64 **ktime_get_tai_fast_ns**(void)

u64 **ktime_get_real_fast_ns**(void)

These variants are safe to call from any context, including from a non-maskable interrupt (NMI) during a timekeeper update, and while we are entering suspend with the clocksource powered down. This is useful in some tracing or debugging code as well as machine check reporting, but most drivers should never call them, since the time is allowed to jump under certain conditions.

2.12.4 Deprecated time interfaces

Older kernels used some other interfaces that are now being phased out but may appear in third-party drivers being ported here. In particular, all interfaces returning a 'struct timeval' or 'struct timespec' have been replaced because the tv_sec member overflows in year 2038 on 32-bit architectures. These are the recommended replacements:

void **ktime_get_ts**(struct timespec*)

Use ktime_get() or ktime_get_ts64() instead.

void **do_gettimeofday**(struct timeval*)

void **getnstimeofday**(struct timespec*)

void **getnstimeofday64**(struct timespec64*)

void **ktime_get_real_ts**(struct timespec*)

ktime_get_real_ts64() is a direct replacement, but consider using monotonic time (ktime_get_ts64()) and/or a ktime_t based interface (ktime_get()/*ktime_get_real()*).

struct timespec **current_kernel_time**(void)

struct timespec64 **current_kernel_time64**(void)

struct timespec **get_monotonic_coarse**(void)

struct timespec64 **get_monotonic_coarse64**(void)

These are replaced by *ktime_get_coarse_real_ts64()* and *ktime_get_coarse_ts64()*. However, A lot of code that wants coarse-grained times can use the simple 'jiffies' instead, while some drivers may actually want the higher resolution accessors these days.

struct timespec **getrawmonotonic**(void)

struct timespec64 **getrawmonotonic64**(void) struct timespec **timekeeping_clocktai**(void)

struct timespec64 **timekeeping_clocktai64**(void)

struct timespec **get_monotonic_boottime**(void)

struct timespec64 **get_monotonic_boottime64**(void)

These are replaced by *ktime_get_raw()*/*ktime_get_raw_ts64()*, *ktime_get_clocktai()*/*ktime_get_clocktai_ts64()* as well as *ktime_get_boottime()*/*ktime_get_boottime_ts64()*. However, if the particular choice of clock source is not important for the user, consider converting to ktime_get()/ktime_get_ts64() instead for consistency.

2.13 The errseq_t datatype

An errseq_t is a way of recording errors in one place, and allowing any number of "subscribers" to tell whether it has changed since a previous point where it was sampled.

The initial use case for this is tracking errors for file synchronization syscalls (fsync, fdatasync, msync and sync_file_range), but it may be usable in other situations.

It's implemented as an unsigned 32-bit value. The low order bits are designated to hold an error code (between 1 and MAX_ERRNO). The upper bits are used as a counter. This is done with atomics instead of locking so that these functions can be called from any context.

Note that there is a risk of collisions if new errors are being recorded frequently, since we have so few bits to use as a counter.

To mitigate this, the bit between the error value and counter is used as a flag to tell whether the value has been sampled since a new value was recorded. That allows us to avoid bumping the counter if no one has sampled it since the last time an error was recorded.

Thus we end up with a value that looks something like this:

31..13	12	11..0
counter	SF	errno

The general idea is for "watchers" to sample an errseq_t value and keep it as a running cursor. That value can later be used to tell whether any new errors have occurred since that sampling was done, and atomically record the state at the time that it was checked. This allows us to record errors in one place, and then have a number of "watchers" that can tell whether the value has changed since they last checked it.

A new errseq_t should always be zeroed out. An errseq_t value of all zeroes is the special (but common) case where there has never been an error. An all zero value thus serves as the "epoch" if one wishes to know whether there has ever been an error set since it was first initialized.

2.13.1 API usage

Let me tell you a story about a worker drone. Now, he's a good worker overall, but the company is a little...management heavy. He has to report to 77 supervisors today, and tomorrow the "big boss" is coming in from out of town and he's sure to test the poor fellow too.

They're all handing him work to do -- so much he can't keep track of who handed him what, but that's not really a big problem. The supervisors just want to know when he's finished all of the work they've handed him so far and whether he made any mistakes since they last asked.

He might have made the mistake on work they didn't actually hand him, but he can't keep track of things at that level of detail, all he can remember is the most recent mistake that he made.

Here's our worker_drone representation:

```
struct worker_drone {
        errseq_t        wd_err; /* for recording errors */
};
```

Every day, the worker_drone starts out with a blank slate:

```
struct worker_drone wd;

wd.wd_err = (errseq_t)0;
```

The supervisors come in and get an initial read for the day. They don't care about anything that happened before their watch begins:

```
struct supervisor {
        errseq_t        s_wd_err; /* private "cursor" for wd_err */
        spinlock_t      s_wd_err_lock; /* protects s_wd_err */
}

struct supervisor       su;

su.s_wd_err = errseq_sample(&wd.wd_err);
spin_lock_init(&su.s_wd_err_lock);
```

Now they start handing him tasks to do. Every few minutes they ask him to finish up all of the work they've handed him so far. Then they ask him whether he made any mistakes on any of it:

```
spin_lock(&su.su_wd_err_lock);
err = errseq_check_and_advance(&wd.wd_err, &su.s_wd_err);
spin_unlock(&su.su_wd_err_lock);
```

Up to this point, that just keeps returning 0.

Now, the owners of this company are quite miserly and have given him substandard equipment with which to do his job. Occasionally it glitches and he makes a mistake. He sighs a heavy sigh, and marks it down:

```
errseq_set(&wd.wd_err, -EIO);
```

...and then gets back to work. The supervisors eventually poll again and they each get the error when they next check. Subsequent calls will return 0, until another error is recorded, at which point it's reported to each of them once.

Note that the supervisors can't tell how many mistakes he made, only whether one was made since they last checked, and the latest value recorded.

Occasionally the big boss comes in for a spot check and asks the worker to do a one-off job for him. He's not really watching the worker full-time like the supervisors, but he does need to know whether a mistake occurred while his job was processing.

He can just sample the current errseq_t in the worker, and then use that to tell whether an error has occurred later:

```
errseq_t since = errseq_sample(&wd.wd_err);
/* submit some work and wait for it to complete */
err = errseq_check(&wd.wd_err, since);
```

Since he's just going to discard "since" after that point, he doesn't need to advance it here. He also doesn't need any locking since it's not usable by anyone else.

2.13.2 Serializing errseq_t cursor updates

Note that the errseq_t API does not protect the errseq_t cursor during a check_and_advance_operation. Only the canonical error code is handled atomically. In a situation where more than one task might be using the same errseq_t cursor at the same time, it's important to serialize updates to that cursor.

If that's not done, then it's possible for the cursor to go backward in which case the same error could be reported more than once.

Because of this, it's often advantageous to first do an errseq_check to see if anything has changed, and only later do an errseq_check_and_advance after taking the lock. e.g.:

```
if (errseq_check(&wd.wd_err, READ_ONCE(su.s_wd_err)) {
        /* su.s_wd_err is protected by s_wd_err_lock */
        spin_lock(&su.s_wd_err_lock);
        err = errseq_check_and_advance(&wd.wd_err, &su.s_wd_err);
        spin_unlock(&su.s_wd_err_lock);
}
```

That avoids the spinlock in the common case where nothing has changed since the last time it was checked.

2.13.3 Functions

errseq_t **errseq_set**(errseq_t *eseq, int err)

set a errseq_t for later reporting

Parameters

errseq_t *eseq

errseq_t field that should be set

int err
error to set (must be between -1 and -MAX_ERRNO)

Description

This function sets the error in **eseq**, and increments the sequence counter if the last sequence was sampled at some point in the past.

Any error set will always overwrite an existing error.

Return

The previous value, primarily for debugging purposes. The return value should not be used as a previously sampled value in later calls as it will not have the SEEN flag set.

errseq_t **errseq_sample**(errseq_t *eseq)
Grab current errseq_t value.

Parameters

errseq_t *eseq
Pointer to errseq_t to be sampled.

Description

This function allows callers to initialise their errseq_t variable. If the error has been "seen", new callers will not see an old error. If there is an unseen error in **eseq**, the caller of this function will see it the next time it checks for an error.

Context

Any context.

Return

The current errseq value.

int **errseq_check**(errseq_t *eseq, errseq_t since)
Has an error occurred since a particular sample point?

Parameters

errseq_t *eseq
Pointer to errseq_t value to be checked.

errseq_t since
Previously-sampled errseq_t from which to check.

Description

Grab the value that eseq points to, and see if it has changed **since** the given value was sampled. The **since** value is not advanced, so there is no need to mark the value as seen.

Return

The latest error set in the errseq_t or 0 if it hasn't changed.

int **errseq_check_and_advance**(errseq_t *eseq, errseq_t *since)
Check an errseq_t and advance to current value.

Parameters

errseq_t *eseq
Pointer to value being checked and reported.

errseq_t *since
Pointer to previously-sampled errseq_t to check against and advance.

Description

Grab the eseq value, and see whether it matches the value that **since** points to. If it does, then just return 0.

If it doesn't, then the value has changed. Set the "seen" flag, and try to swap it into place as the new eseq value. Then, set that value as the new "since" value, and return whatever the error portion is set to.

Note that no locking is provided here for concurrent updates to the "since" value. The caller must provide that if necessary. Because of this, callers may want to do a lockless errseq_check before taking the lock and calling this.

Return

Negative errno if one has been stored, or 0 if no new error has occurred.

2.14 Atomic types

```
On atomic types (atomic_t atomic64_t and atomic_long_t).

The atomic type provides an interface to the architecture's means of atomic
RMW operations between CPUs (atomic operations on MMIO are not supported and
can lead to fatal traps on some platforms).

API
---

The 'full' API consists of (atomic64_ and atomic_long_ prefixes omitted for
brevity):

Non-RMW ops:

  atomic_read(), atomic_set()
  atomic_read_acquire(), atomic_set_release()


RMW atomic operations:

Arithmetic:

  atomic_{add,sub,inc,dec}()
  atomic_{add,sub,inc,dec}_return{,_relaxed,_acquire,_release}()
  atomic_fetch_{add,sub,inc,dec}{,_relaxed,_acquire,_release}()


Bitwise:

  atomic_{and,or,xor,andnot}()
  atomic_fetch_{and,or,xor,andnot}{,_relaxed,_acquire,_release}()


Swap:
```

```
  atomic_xchg{,_relaxed,_acquire,_release}()
  atomic_cmpxchg{,_relaxed,_acquire,_release}()
  atomic_try_cmpxchg{,_relaxed,_acquire,_release}()


Reference count (but please see refcount_t):

  atomic_add_unless(), atomic_inc_not_zero()
  atomic_sub_and_test(), atomic_dec_and_test()


Misc:

  atomic_inc_and_test(), atomic_add_negative()
  atomic_dec_unless_positive(), atomic_inc_unless_negative()


Barriers:

  smp_mb__{before,after}_atomic()


TYPES (signed vs unsigned)
-----

While atomic_t, atomic_long_t and atomic64_t use int, long and s64
respectively (for hysterical raisins), the kernel uses -fno-strict-overflow
(which implies -fwrapv) and defines signed overflow to behave like
2s-complement.

Therefore, an explicitly unsigned variant of the atomic ops is strictly
unnecessary and we can simply cast, there is no UB.

There was a bug in UBSAN prior to GCC-8 that would generate UB warnings for
signed types.

With this we also conform to the C/C++ _Atomic behaviour and things like
P1236R1.


SEMANTICS
---------

Non-RMW ops:

The non-RMW ops are (typically) regular LOADs and STOREs and are canonically
implemented using READ_ONCE(), WRITE_ONCE(), smp_load_acquire() and
smp_store_release() respectively. Therefore, if you find yourself only using
the Non-RMW operations of atomic_t, you do not in fact need atomic_t at all
and are doing it wrong.

A note for the implementation of atomic_set{}() is that it must not break the
atomicity of the RMW ops. That is:

  C Atomic-RMW-ops-are-atomic-WRT-atomic_set

  {
    atomic_t v = ATOMIC_INIT(1);
  }

  P0(atomic_t *v)
  {
```

```
    (void)atomic_add_unless(v, 1, 0);
  }

  P1(atomic_t *v)
  {
    atomic_set(v, 0);
  }

  exists
  (v=2)

In this case we would expect the atomic_set() from CPU1 to either happen
before the atomic_add_unless(), in which case that latter one would no-op, or
_after_ in which case we'd overwrite its result. In no case is "2" a valid
outcome.

This is typically true on 'normal' platforms, where a regular competing STORE
will invalidate a LL/SC or fail a CMPXCHG.

The obvious case where this is not so is when we need to implement atomic ops
with a lock:

  CPU0                                          CPU1

  atomic_add_unless(v, 1, 0);
    lock();
    ret = READ_ONCE(v->counter); // == 1
                                                atomic_set(v, 0);
    if (ret != u)                                 WRITE_ONCE(v->counter, 0);
      WRITE_ONCE(v->counter, ret + 1);
    unlock();

the typical solution is to then implement atomic_set{}() with atomic_xchg().


RMW ops:

These come in various forms:

 - plain operations without return value: atomic_{}()

 - operations which return the modified value: atomic_{}_return()

   these are limited to the arithmetic operations because those are
   reversible. Bitops are irreversible and therefore the modified value
   is of dubious utility.

 - operations which return the original value: atomic_fetch_{}()

 - swap operations: xchg(), cmpxchg() and try_cmpxchg()

 - misc; the special purpose operations that are commonly used and would,
   given the interface, normally be implemented using (try_)cmpxchg loops but
   are time critical and can, (typically) on LL/SC architectures, be more
   efficiently implemented.

All these operations are SMP atomic; that is, the operations (for a single
atomic variable) can be fully ordered and no intermediate state is lost or
visible.


ORDERING  (go read memory-barriers.txt first)
--------
```

```
The rule of thumb:

 - non-RMW operations are unordered;

 - RMW operations that have no return value are unordered;

 - RMW operations that have a return value are fully ordered;

 - RMW operations that are conditional are unordered on FAILURE,
   otherwise the above rules apply.

Except of course when an operation has an explicit ordering like:

 {}_relaxed: unordered
 {}_acquire: the R of the RMW (or atomic_read) is an ACQUIRE
 {}_release: the W of the RMW (or atomic_set)  is a  RELEASE

Where 'unordered' is against other memory locations. Address dependencies are
not defeated.

Fully ordered primitives are ordered against everything prior and everything
subsequent. Therefore a fully ordered primitive is like having an smp_mb()
before and an smp_mb() after the primitive.


The barriers:

  smp_mb__{before,after}_atomic()

only apply to the RMW atomic ops and can be used to augment/upgrade the
ordering inherent to the op. These barriers act almost like a full smp_mb():
smp_mb__before_atomic() orders all earlier accesses against the RMW op
itself and all accesses following it, and smp_mb__after_atomic() orders all
later accesses against the RMW op and all accesses preceding it. However,
accesses between the smp_mb__{before,after}_atomic() and the RMW op are not
ordered, so it is advisable to place the barrier right next to the RMW atomic
op whenever possible.

These helper barriers exist because architectures have varying implicit
ordering on their SMP atomic primitives. For example our TSO architectures
provide full ordered atomics and these barriers are no-ops.

NOTE: when the atomic RmW ops are fully ordered, they should also imply a
compiler barrier.

Thus:

  atomic_fetch_add();

is equivalent to:

  smp_mb__before_atomic();
  atomic_fetch_add_relaxed();
  smp_mb__after_atomic();

However the atomic_fetch_add() might be implemented more efficiently.

Further, while something like:

  smp_mb__before_atomic();
  atomic_dec(&X);
```

```
is a 'typical' RELEASE pattern, the barrier is strictly stronger than
a RELEASE because it orders preceding instructions against both the read
and write parts of the atomic_dec(), and against all following instructions
as well. Similarly, something like:

  atomic_inc(&X);
  smp_mb__after_atomic();

is an ACQUIRE pattern (though very much not typical), but again the barrier is
strictly stronger than ACQUIRE. As illustrated:

  C Atomic-RMW+mb__after_atomic-is-stronger-than-acquire

  {
  }

  P0(int *x, atomic_t *y)
  {
    r0 = READ_ONCE(*x);
    smp_rmb();
    r1 = atomic_read(y);
  }

  P1(int *x, atomic_t *y)
  {
    atomic_inc(y);
    smp_mb__after_atomic();
    WRITE_ONCE(*x, 1);
  }

  exists
  (0:r0=1 /\ 0:r1=0)

This should not happen; but a hypothetical atomic_inc_acquire() --
(void)atomic_fetch_inc_acquire() for instance -- would allow the outcome,
because it would not order the W part of the RMW against the following
WRITE_ONCE.  Thus:

  P0                    P1

                        t = LL.acq *y (0)
                        t++;
                        *x = 1;
  r0 = *x (1)
  RMB
  r1 = *y (0)
                        SC *y, t;

is allowed.


CMPXCHG vs TRY_CMPXCHG
----------------------

  int atomic_cmpxchg(atomic_t *ptr, int old, int new);
  bool atomic_try_cmpxchg(atomic_t *ptr, int *oldp, int new);

Both provide the same functionality, but try_cmpxchg() can lead to more
compact code. The functions relate like:

  bool atomic_try_cmpxchg(atomic_t *ptr, int *oldp, int new)
  {
    int ret, old = *oldp;
```

```
    ret = atomic_cmpxchg(ptr, old, new);
    if (ret != old)
      *oldp = ret;
    return ret == old;
  }
```

and:

```
  int atomic_cmpxchg(atomic_t *ptr, int old, int new)
  {
    (void)atomic_try_cmpxchg(ptr, &old, new);
    return old;
  }
```

Usage:

```
  old = atomic_read(&v);                         old = atomic_read(&v);
  for (;;) {                                     do {
    new = func(old);                               new = func(old);
    tmp = atomic_cmpxchg(&v, old, new);          } while (!atomic_try_cmpxchg(&v, &old, new));
    if (tmp == old)
      break;
    old = tmp;
  }
```

NB. try_cmpxchg() also generates better code on some platforms (notably x86) where the function more closely matches the hardware instruction.

FORWARD PROGRESS

In general strong forward progress is expected of all unconditional atomic operations -- those in the Arithmetic and Bitwise classes and xchg(). However a fair amount of code also requires forward progress from the conditional atomic operations.

Specifically 'simple' cmpxchg() loops are expected to not starve one another indefinitely. However, this is not evident on LL/SC architectures, because while an LL/SC architecture 'can/should/must' provide forward progress guarantees between competing LL/SC sections, such a guarantee does not transfer to cmpxchg() implemented using LL/SC. Consider:

```
  old = atomic_read(&v);
  do {
    new = func(old);
  } while (!atomic_try_cmpxchg(&v, &old, new));
```

which on LL/SC becomes something like:

```
  old = atomic_read(&v);
  do {
    new = func(old);
  } while (!({
    volatile asm ("1: LL  %[oldval], %[v]\n"
                  "   CMP %[oldval], %[old]\n"
                  "   BNE 2f\n"
                  "   SC  %[new], %[v]\n"
                  "   BNE 1b\n"
                  "2:\n"
                  : [oldval] "=&r" (oldval), [v] "m" (v)
                  : [old] "r" (old), [new] "r" (new)
                  : "memory");
```

```
    success = (oldval == old);
    if (!success)
      old = oldval;
    success; }));

However, even the forward branch from the failed compare can cause the LL/SC
to fail on some architectures, let alone whatever the compiler makes of the C
loop body. As a result there is no guarantee what so ever the cacheline
containing @v will stay on the local CPU and progress is made.

Even native CAS architectures can fail to provide forward progress for their
primitive (See Sparc64 for an example).

Such implementations are strongly encouraged to add exponential backoff loops
to a failed CAS in order to ensure some progress. Affected architectures are
also strongly encouraged to inspect/audit the atomic fallbacks, refcount_t and
their locking primitives.
```

2.15 Atomic bitops

```
=============
Atomic bitops
=============

While our bitmap_{}() functions are non-atomic, we have a number of operations
operating on single bits in a bitmap that are atomic.


API
---

The single bit operations are:

Non-RMW ops:

  test_bit()

RMW atomic operations without return value:

  {set,clear,change}_bit()
  clear_bit_unlock()

RMW atomic operations with return value:

  test_and_{set,clear,change}_bit()
  test_and_set_bit_lock()

Barriers:

  smp_mb__{before,after}_atomic()


All RMW atomic operations have a '__' prefixed variant which is non-atomic.


SEMANTICS
---------

Non-atomic ops:
```

```
In particular __clear_bit_unlock() suffers the same issue as atomic_set(),
which is why the generic version maps to clear_bit_unlock(), see atomic_t.txt.


RMW ops:

The test_and_{}_bit() operations return the original value of the bit.


ORDERING
--------

Like with atomic_t, the rule of thumb is:

 - non-RMW operations are unordered;

 - RMW operations that have no return value are unordered;

 - RMW operations that have a return value are fully ordered.

 - RMW operations that are conditional are fully ordered.

Except for a successful test_and_set_bit_lock() which has ACQUIRE semantics,
clear_bit_unlock() which has RELEASE semantics and test_bit_acquire which has
ACQUIRE semantics.

Since a platform only has a single means of achieving atomic operations
the same barriers as for atomic_t are used, see atomic_t.txt.
```

CHAPTER THREE

LOW LEVEL ENTRY AND EXIT

3.1 Entry/exit handling for exceptions, interrupts, syscalls and KVM

All transitions between execution domains require state updates which are subject to strict ordering constraints. State updates are required for the following:

- Lockdep
- RCU / Context tracking
- Preemption counter
- Tracing
- Time accounting

The update order depends on the transition type and is explained below in the transition type sections: *Syscalls*, *KVM*, *Interrupts and regular exceptions*, *NMI and NMI-like exceptions*.

3.1.1 Non-instrumentable code - noinstr

Most instrumentation facilities depend on RCU, so intrumentation is prohibited for entry code before RCU starts watching and exit code after RCU stops watching. In addition, many architectures must save and restore register state, which means that (for example) a breakpoint in the breakpoint entry code would overwrite the debug registers of the initial breakpoint.

Such code must be marked with the 'noinstr' attribute, placing that code into a special section inaccessible to instrumentation and debug facilities. Some functions are partially instrumentable, which is handled by marking them noinstr and using instrumentation_begin() and instrumentation_end() to flag the instrumentable ranges of code:

```
noinstr void entry(void)
{
      handle_entry();     // <-- must be 'noinstr' or '__always_inline'
      ...

      instrumentation_begin();
      handle_context();   // <-- instrumentable code
      instrumentation_end();

      ...
```

```
        handle_exit();      // <-- must be 'noinstr' or '__always_inline'
}
```

This allows verification of the 'noinstr' restrictions via objtool on supported architectures.

Invoking non-instrumentable functions from instrumentable context has no restrictions and is useful to protect e.g. state switching which would cause malfunction if instrumented.

All non-instrumentable entry/exit code sections before and after the RCU state transitions must run with interrupts disabled.

3.1.2 Syscalls

Syscall-entry code starts in assembly code and calls out into low-level C code after establishing low-level architecture-specific state and stack frames. This low-level C code must not be instrumented. A typical syscall handling function invoked from low-level assembly code looks like this:

```
noinstr void syscall(struct pt_regs *regs, int nr)
{
        arch_syscall_enter(regs);
        nr = syscall_enter_from_user_mode(regs, nr);

        instrumentation_begin();
        if (!invoke_syscall(regs, nr) && nr != -1)
                result_reg(regs) = __sys_ni_syscall(regs);
        instrumentation_end();

        syscall_exit_to_user_mode(regs);
}
```

syscall_enter_from_user_mode() first invokes enter_from_user_mode() which establishes state in the following order:

- Lockdep
- RCU / Context tracking
- Tracing

and then invokes the various entry work functions like ptrace, seccomp, audit, syscall tracing, etc. After all that is done, the instrumentable invoke_syscall function can be invoked. The instrumentable code section then ends, after which syscall_exit_to_user_mode() is invoked.

syscall_exit_to_user_mode() handles all work which needs to be done before returning to user space like tracing, audit, signals, task work etc. After that it invokes exit_to_user_mode() which again handles the state transition in the reverse order:

- Tracing
- RCU / Context tracking
- Lockdep

syscall_enter_from_user_mode() and syscall_exit_to_user_mode() are also available as fine grained subfunctions in cases where the architecture code has to do extra work between the various steps. In such cases it has to ensure that enter_from_user_mode() is called first on entry and exit_to_user_mode() is called last on exit.

Do not nest syscalls. Nested systcalls will cause RCU and/or context tracking to print a warning.

3.1.3 KVM

Entering or exiting guest mode is very similar to syscalls. From the host kernel point of view the CPU goes off into user space when entering the guest and returns to the kernel on exit.

kvm_guest_enter_irqoff() is a KVM-specific variant of exit_to_user_mode() and kvm_guest_exit_irqoff() is the KVM variant of enter_from_user_mode(). The state operations have the same ordering.

Task work handling is done separately for guest at the boundary of the vcpu_run() loop via xfer_to_guest_mode_handle_work() which is a subset of the work handled on return to user space.

Do not nest KVM entry/exit transitions because doing so is nonsensical.

3.1.4 Interrupts and regular exceptions

Interrupts entry and exit handling is slightly more complex than syscalls and KVM transitions.

If an interrupt is raised while the CPU executes in user space, the entry and exit handling is exactly the same as for syscalls.

If the interrupt is raised while the CPU executes in kernel space the entry and exit handling is slightly different. RCU state is only updated when the interrupt is raised in the context of the CPU's idle task. Otherwise, RCU will already be watching. Lockdep and tracing have to be updated unconditionally.

irqentry_enter() and irqentry_exit() provide the implementation for this.

The architecture-specific part looks similar to syscall handling:

```
noinstr void interrupt(struct pt_regs *regs, int nr)
{
      arch_interrupt_enter(regs);
      state = irqentry_enter(regs);

      instrumentation_begin();

      irq_enter_rcu();
      invoke_irq_handler(regs, nr);
      irq_exit_rcu();

      instrumentation_end();

      irqentry_exit(regs, state);
}
```

Note that the invocation of the actual interrupt handler is within a irq_enter_rcu() and irq_exit_rcu() pair.

irq_enter_rcu() updates the preemption count which makes in_hardirq() return true, handles NOHZ tick state and interrupt time accounting. This means that up to the point where irq_enter_rcu() is invoked in_hardirq() returns false.

irq_exit_rcu() handles interrupt time accounting, undoes the preemption count update and eventually handles soft interrupts and NOHZ tick state.

In theory, the preemption count could be updated in irqentry_enter(). In practice, deferring this update to irq_enter_rcu() allows the preemption-count code to be traced, while also maintaining symmetry with irq_exit_rcu() and irqentry_exit(), which are described in the next paragraph. The only downside is that the early entry code up to irq_enter_rcu() must be aware that the preemption count has not yet been updated with the HARDIRQ_OFFSET state.

Note that irq_exit_rcu() must remove HARDIRQ_OFFSET from the preemption count before it handles soft interrupts, whose handlers must run in BH context rather than irq-disabled context. In addition, irqentry_exit() might schedule, which also requires that HARDIRQ_OFFSET has been removed from the preemption count.

Even though interrupt handlers are expected to run with local interrupts disabled, interrupt nesting is common from an entry/exit perspective. For example, softirq handling happens within an irqentry_{enter,exit}() block with local interrupts enabled. Also, although uncommon, nothing prevents an interrupt handler from re-enabling interrupts.

Interrupt entry/exit code doesn't strictly need to handle reentrancy, since it runs with local interrupts disabled. But NMIs can happen anytime, and a lot of the entry code is shared between the two.

3.1.5 NMI and NMI-like exceptions

NMIs and NMI-like exceptions (machine checks, double faults, debug interrupts, etc.) can hit any context and must be extra careful with the state.

State changes for debug exceptions and machine-check exceptions depend on whether these exceptions happened in user-space (breakpoints or watchpoints) or in kernel mode (code patching). From user-space, they are treated like interrupts, while from kernel mode they are treated like NMIs.

NMIs and other NMI-like exceptions handle state transitions without distinguishing between user-mode and kernel-mode origin.

The state update on entry is handled in irqentry_nmi_enter() which updates state in the following order:

- Preemption counter
- Lockdep
- RCU / Context tracking
- Tracing

The exit counterpart irqentry_nmi_exit() does the reverse operation in the reverse order.

Note that the update of the preemption counter has to be the first operation on enter and the last operation on exit. The reason is that both lockdep and RCU rely on in_nmi() returning true in this case. The preemption count modification in the NMI entry/exit case must not be traced.

Architecture-specific code looks like this:

```
noinstr void nmi(struct pt_regs *regs)
{
	arch_nmi_enter(regs);
	state = irqentry_nmi_enter(regs);

	instrumentation_begin();
	nmi_handler(regs);
	instrumentation_end();

	irqentry_nmi_exit(regs);
}
```

and for e.g. a debug exception it can look like this:

```
noinstr void debug(struct pt_regs *regs)
{
	arch_nmi_enter(regs);

	debug_regs = save_debug_regs();

	if (user_mode(regs)) {
		state = irqentry_enter(regs);

		instrumentation_begin();
		user_mode_debug_handler(regs, debug_regs);
		instrumentation_end();

		irqentry_exit(regs, state);
	} else {
		state = irqentry_nmi_enter(regs);

		instrumentation_begin();
		kernel_mode_debug_handler(regs, debug_regs);
		instrumentation_end();

		irqentry_nmi_exit(regs, state);
	}
}
```

There is no combined irqentry_nmi_if_kernel() function available as the above cannot be handled in an exception-agnostic way.

NMIs can happen in any context. For example, an NMI-like exception triggered while handling an NMI. So NMI entry code has to be reentrant and state updates need to handle nesting.

CHAPTER

FOUR

CONCURRENCY PRIMITIVES

How Linux keeps everything from happening at the same time. See Documentation/locking/index.rst for more related documentation.

4.1 refcount_t API compared to atomic_t

4.1.1 Introduction

The goal of refcount_t API is to provide a minimal API for implementing an object's reference counters. While a generic architecture-independent implementation from lib/refcount.c uses atomic operations underneath, there are a number of differences between some of the `refcount_*()` and `atomic_*()` functions with regards to the memory ordering guarantees. This document outlines the differences and provides respective examples in order to help maintainers validate their code against the change in these memory ordering guarantees.

The terms used through this document try to follow the formal LKMM defined in tools/memory-model/Documentation/explanation.txt.

memory-barriers.txt and atomic_t.txt provide more background to the memory ordering in general and for atomic operations specifically.

4.1.2 Relevant types of memory ordering

Note: The following section only covers some of the memory ordering types that are relevant for the atomics and reference counters and used through this document. For a much broader picture please consult memory-barriers.txt document.

In the absence of any memory ordering guarantees (i.e. fully unordered) atomics & refcounters only provide atomicity and program order (po) relation (on the same CPU). It guarantees that each `atomic_*()` and `refcount_*()` operation is atomic and instructions are executed in program order on a single CPU. This is implemented using READ_ONCE()/WRITE_ONCE() and compare-and-swap primitives.

A strong (full) memory ordering guarantees that all prior loads and stores (all po-earlier instructions) on the same CPU are completed before any po-later instruction is executed on the same CPU. It also guarantees that all po-earlier stores on the same CPU and all propagated stores from other CPUs must propagate to all other CPUs before any po-later instruction is executed on the original CPU (A-cumulative property). This is implemented using smp_mb().

A RELEASE memory ordering guarantees that all prior loads and stores (all po-earlier instructions) on the same CPU are completed before the operation. It also guarantees that all po-earlier stores on the same CPU and all propagated stores from other CPUs must propagate to all other CPUs before the release operation (A-cumulative property). This is implemented using smp_store_release().

An ACQUIRE memory ordering guarantees that all post loads and stores (all po-later instructions) on the same CPU are completed after the acquire operation. It also guarantees that all po-later stores on the same CPU must propagate to all other CPUs after the acquire operation executes. This is implemented using smp_acquire__after_ctrl_dep().

A control dependency (on success) for refcounters guarantees that if a reference for an object was successfully obtained (reference counter increment or addition happened, function returned true), then further stores are ordered against this operation. Control dependency on stores are not implemented using any explicit barriers, but rely on CPU not to speculate on stores. This is only a single CPU relation and provides no guarantees for other CPUs.

4.1.3 Comparison of functions

case 1) - non-"Read/Modify/Write" (RMW) ops

Function changes:

- atomic_set() --> refcount_set()
- atomic_read() --> refcount_read()

Memory ordering guarantee changes:

- none (both fully unordered)

case 2) - increment-based ops that return no value

Function changes:

- atomic_inc() --> refcount_inc()
- atomic_add() --> refcount_add()

Memory ordering guarantee changes:

- none (both fully unordered)

case 3) - decrement-based RMW ops that return no value

Function changes:

- atomic_dec() --> refcount_dec()

Memory ordering guarantee changes:

- fully unordered --> RELEASE ordering

case 4) - increment-based RMW ops that return a value

Function changes:

- atomic_inc_not_zero() --> refcount_inc_not_zero()
- no atomic counterpart --> refcount_add_not_zero()

Memory ordering guarantees changes:

- fully ordered --> control dependency on success for stores

Note: We really assume here that necessary ordering is provided as a result of obtaining pointer to the object!

case 5) - generic dec/sub decrement-based RMW ops that return a value

Function changes:

- atomic_dec_and_test() --> refcount_dec_and_test()
- atomic_sub_and_test() --> refcount_sub_and_test()

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + ACQUIRE ordering on success

case 6) other decrement-based RMW ops that return a value

Function changes:

- no atomic counterpart --> refcount_dec_if_one()
- `atomic_add_unless(&var, -1, 1)` --> `refcount_dec_not_one(&var)`

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + control dependency

Note: atomic_add_unless() only provides full order on success.

case 7) - lock-based RMW

Function changes:

- atomic_dec_and_lock() --> refcount_dec_and_lock()
- atomic_dec_and_mutex_lock() --> refcount_dec_and_mutex_lock()

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + control dependency + hold spin_lock() on success

4.2 IRQs

4.2.1 What is an IRQ?

An IRQ is an interrupt request from a device. Currently they can come in over a pin, or over a packet. Several devices may be connected to the same pin thus sharing an IRQ.

An IRQ number is a kernel identifier used to talk about a hardware interrupt source. Typically this is an index into the global irq_desc array, but except for what linux/interrupt.h implements the details are architecture specific.

An IRQ number is an enumeration of the possible interrupt sources on a machine. Typically what is enumerated is the number of input pins on all of the interrupt controller in the system. In the case of ISA what is enumerated are the 16 input pins on the two i8259 interrupt controllers.

Architectures can assign additional meaning to the IRQ numbers, and are encouraged to in the case where there is any manual configuration of the hardware involved. The ISA IRQs are a classic example of assigning this kind of additional meaning.

4.2.2 SMP IRQ affinity

ChangeLog:

- Started by Ingo Molnar <mingo@redhat.com>
- Update by Max Krasnyansky <maxk@qualcomm.com>

/proc/irq/IRQ#/smp_affinity and /proc/irq/IRQ#/smp_affinity_list specify which target CPUs are permitted for a given IRQ source. It's a bitmask (smp_affinity) or cpu list (smp_affinity_list) of allowed CPUs. It's not allowed to turn off all CPUs, and if an IRQ controller does not support IRQ affinity then the value will not change from the default of all cpus.

/proc/irq/default_smp_affinity specifies default affinity mask that applies to all non-active IRQs. Once IRQ is allocated/activated its affinity bitmask will be set to the default mask. It can then be changed as described above. Default mask is 0xffffffff.

Here is an example of restricting IRQ44 (eth1) to CPU0-3 then restricting it to CPU4-7 (this is an 8-CPU SMP box):

```
[root@moon 44]# cd /proc/irq/44
[root@moon 44]# cat smp_affinity
ffffffff

[root@moon 44]# echo 0f > smp_affinity
[root@moon 44]# cat smp_affinity
0000000f
[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
...
--- hell ping statistics ---
6029 packets transmitted, 6027 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.4 ms
[root@moon 44]# cat /proc/interrupts | grep 'CPU\|44:'
        CPU0       CPU1       CPU2       CPU3      CPU4       CPU5        CPU6␣
↪       CPU7
44:       1068       1785       1785       1783         0          0           ␣
↪0         0    IO-APIC-level  eth1
```

As can be seen from the line above IRQ44 was delivered only to the first four processors (0-3). Now lets restrict that IRQ to CPU(4-7).

```
[root@moon 44]# echo f0 > smp_affinity
[root@moon 44]# cat smp_affinity
000000f0
[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
..
--- hell ping statistics ---
2779 packets transmitted, 2777 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.5/585.4 ms
[root@moon 44]# cat /proc/interrupts |  'CPU\|44:'
        CPU0       CPU1       CPU2       CPU3      CPU4       CPU5        CPU6␣
↪       CPU7
```

```
44:       1068       1785       1785       1783       1784       1069      ␣
↪1070       1069   IO-APIC-level  eth1
```

This time around IRQ44 was delivered only to the last four processors. i.e counters for the CPU0-3 did not change.

Here is an example of limiting that same irq (44) to cpus 1024 to 1031:

```
[root@moon 44]# echo 1024-1031 > smp_affinity_list
[root@moon 44]# cat smp_affinity_list
1024-1031
```

Note that to do this with a bitmask would require 32 bitmasks of zero to follow the pertinent one.

4.2.3 The irq_domain interrupt number mapping library

The current design of the Linux kernel uses a single large number space where each separate IRQ source is assigned a different number. This is simple when there is only one interrupt controller, but in systems with multiple interrupt controllers the kernel must ensure that each one gets assigned non-overlapping allocations of Linux IRQ numbers.

The number of interrupt controllers registered as unique irqchips show a rising tendency: for example subdrivers of different kinds such as GPIO controllers avoid reimplementing identical callback mechanisms as the IRQ core system by modelling their interrupt handlers as irqchips, i.e. in effect cascading interrupt controllers.

Here the interrupt number loose all kind of correspondence to hardware interrupt numbers: whereas in the past, IRQ numbers could be chosen so they matched the hardware IRQ line into the root interrupt controller (i.e. the component actually fireing the interrupt line to the CPU) nowadays this number is just a number.

For this reason we need a mechanism to separate controller-local interrupt numbers, called hardware irq's, from Linux IRQ numbers.

The irq_alloc_desc*() and irq_free_desc*() APIs provide allocation of irq numbers, but they don't provide any support for reverse mapping of the controller-local IRQ (hwirq) number into the Linux IRQ number space.

The irq_domain library adds mapping between hwirq and IRQ numbers on top of the irq_alloc_desc*() API. An irq_domain to manage mapping is preferred over interrupt controller drivers open coding their own reverse mapping scheme.

irq_domain also implements translation from an abstract irq_fwspec structure to hwirq numbers (Device Tree and ACPI GSI so far), and can be easily extended to support other IRQ topology data sources.

irq_domain usage

An interrupt controller driver creates and registers an irq_domain by calling one of the irq_domain_add_*() or irq_domain_create_*() functions (each mapping method has a different allocator function, more on that later). The function will return a pointer to the irq_domain on success. The caller must provide the allocator function with an irq_domain_ops structure.

In most cases, the irq_domain will begin empty without any mappings between hwirq and IRQ numbers. Mappings are added to the irq_domain by calling irq_create_mapping() which accepts the irq_domain and a hwirq number as arguments. If a mapping for the hwirq doesn't already exist then it will allocate a new Linux irq_desc, associate it with the hwirq, and call the .map() callback so the driver can perform any required hardware setup.

Once a mapping has been established, it can be retrieved or used via a variety of methods:

- irq_resolve_mapping() returns a pointer to the irq_desc structure for a given domain and hwirq number, and NULL if there was no mapping.
- irq_find_mapping() returns a Linux IRQ number for a given domain and hwirq number, and 0 if there was no mapping
- irq_linear_revmap() is now identical to irq_find_mapping(), and is deprecated
- generic_handle_domain_irq() handles an interrupt described by a domain and a hwirq number

Note that irq domain lookups must happen in contexts that are compatible with a RCU read-side critical section.

The irq_create_mapping() function must be called *at least once* before any call to irq_find_mapping(), lest the descriptor will not be allocated.

If the driver has the Linux IRQ number or the irq_data pointer, and needs to know the associated hwirq number (such as in the irq_chip callbacks) then it can be directly obtained from irq_data->hwirq.

Types of irq_domain mappings

There are several mechanisms available for reverse mapping from hwirq to Linux irq, and each mechanism uses a different allocation function. Which reverse map type should be used depends on the use case. Each of the reverse map types are described below:

Linear

```
irq_domain_add_linear()
irq_domain_create_linear()
```

The linear reverse map maintains a fixed size table indexed by the hwirq number. When a hwirq is mapped, an irq_desc is allocated for the hwirq, and the IRQ number is stored in the table.

The Linear map is a good choice when the maximum number of hwirqs is fixed and a relatively small number (~ < 256). The advantages of this map are fixed time lookup for IRQ numbers, and irq_descs are only allocated for in-use IRQs. The disadvantage is that the table must be as large as the largest possible hwirq number.

irq_domain_add_linear() and irq_domain_create_linear() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

The majority of drivers should use the linear map.

Tree

```
irq_domain_add_tree()
irq_domain_create_tree()
```

The irq_domain maintains a radix tree map from hwirq numbers to Linux IRQs. When an hwirq is mapped, an irq_desc is allocated and the hwirq is used as the lookup key for the radix tree.

The tree map is a good choice if the hwirq number can be very large since it doesn't need to allocate a table as large as the largest hwirq number. The disadvantage is that hwirq to IRQ number lookup is dependent on how many entries are in the table.

irq_domain_add_tree() and irq_domain_create_tree() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

Very few drivers should need this mapping.

No Map

```
irq_domain_add_nomap()
```

The No Map mapping is to be used when the hwirq number is programmable in the hardware. In this case it is best to program the Linux IRQ number into the hardware itself so that no mapping is required. Calling irq_create_direct_mapping() will allocate a Linux IRQ number and call the .map() callback so that driver can program the Linux IRQ number into the hardware.

Most drivers cannot use this mapping, and it is now gated on the CONFIG_IRQ_DOMAIN_NOMAP option. Please refrain from introducing new users of this API.

Legacy

```
irq_domain_add_simple()
irq_domain_add_legacy()
irq_domain_create_simple()
irq_domain_create_legacy()
```

The Legacy mapping is a special case for drivers that already have a range of irq_descs allocated for the hwirqs. It is used when the driver cannot be immediately converted to use the linear mapping. For example, many embedded system board support files use a set of #defines for IRQ numbers that are passed to struct device registrations. In that case the Linux IRQ numbers cannot be dynamically assigned and the legacy mapping should be used.

As the name implies, the *_legacy() functions are deprecated and only exist to ease the support of ancient platforms. No new users should be added. Same goes for the *_simple() functions when their use results in the legacy behaviour.

The legacy map assumes a contiguous range of IRQ numbers has already been allocated for the controller and that the IRQ number can be calculated by adding a fixed offset to the hwirq number, and visa-versa. The disadvantage is that it requires the interrupt controller to manage IRQ allocations and it requires an irq_desc to be allocated for every hwirq, even if it is unused.

The legacy map should only be used if fixed IRQ mappings must be supported. For example, ISA controllers would use the legacy map for mapping Linux IRQs 0-15 so that existing ISA drivers get the correct IRQ numbers.

Most users of legacy mappings should use irq_domain_add_simple() or irq_domain_create_simple() which will use a legacy domain only if an IRQ range is supplied by the system and will otherwise use a linear domain mapping. The semantics of this call are such that if an IRQ range is specified then descriptors will be allocated on-the-fly for it, and if no range is specified it will fall through to irq_domain_add_linear() or irq_domain_create_linear() which means *no* irq descriptors will be allocated.

A typical use case for simple domains is where an irqchip provider is supporting both dynamic and static IRQ assignments.

In order to avoid ending up in a situation where a linear domain is used and no descriptor gets allocated it is very important to make sure that the driver using the simple domain call irq_create_mapping() before any irq_find_mapping() since the latter will actually work for the static IRQ assignment case.

irq_domain_add_simple() and irq_domain_create_simple() as well as irq_domain_add_legacy() and irq_domain_create_legacy() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

Hierarchy IRQ domain

On some architectures, there may be multiple interrupt controllers involved in delivering an interrupt from the device to the target CPU. Let's look at a typical interrupt delivering path on x86 platforms:

```
Device --> IOAPIC -> Interrupt remapping Controller -> Local APIC -> CPU
```

There are three interrupt controllers involved:

1) IOAPIC controller
2) Interrupt remapping controller
3) Local APIC controller

To support such a hardware topology and make software architecture match hardware architecture, an irq_domain data structure is built for each interrupt controller and those irq_domains are organized into hierarchy. When building irq_domain hierarchy, the irq_domain near to the device is child and the irq_domain near to CPU is parent. So a hierarchy structure as below will be built for the example above:

```
CPU Vector irq_domain (root irq_domain to manage CPU vectors)
        ^
        |
Interrupt Remapping irq_domain (manage irq_remapping entries)
        ^
        |
IOAPIC irq_domain (manage IOAPIC delivery entries/pins)
```

There are four major interfaces to use hierarchy irq_domain:

1) irq_domain_alloc_irqs(): allocate IRQ descriptors and interrupt controller related resources to deliver these interrupts.
2) irq_domain_free_irqs(): free IRQ descriptors and interrupt controller related resources associated with these interrupts.
3) irq_domain_activate_irq(): activate interrupt controller hardware to deliver the interrupt.
4) irq_domain_deactivate_irq(): deactivate interrupt controller hardware to stop delivering the interrupt.

Following changes are needed to support hierarchy irq_domain:

1) a new field 'parent' is added to struct irq_domain; it's used to maintain irq_domain hierarchy information.
2) a new field 'parent_data' is added to *struct irq_data*; it's used to build hierarchy irq_data to match hierarchy irq_domains. The irq_data is used to store irq_domain pointer and hardware irq number.
3) new callbacks are added to struct irq_domain_ops to support hierarchy irq_domain operations.

With support of hierarchy irq_domain and hierarchy irq_data ready, an irq_domain structure is built for each interrupt controller, and an irq_data structure is allocated for each irq_domain associated with an IRQ. Now we could go one step further to support stacked(hierarchy) irq_chip. That is, an irq_chip is associated with each irq_data along the hierarchy. A child irq_chip may implement a required action by itself or by cooperating with its parent irq_chip.

With stacked irq_chip, interrupt controller driver only needs to deal with the hardware managed by itself and may ask for services from its parent irq_chip when needed. So we could achieve a much cleaner software architecture.

For an interrupt controller driver to support hierarchy irq_domain, it needs to:

1) Implement irq_domain_ops.alloc and irq_domain_ops.free
2) Optionally implement irq_domain_ops.activate and irq_domain_ops.deactivate.
3) Optionally implement an irq_chip to manage the interrupt controller hardware.
4) No need to implement irq_domain_ops.map and irq_domain_ops.unmap, they are unused with hierarchy irq_domain.

Hierarchy irq_domain is in no way x86 specific, and is heavily used to support other architectures, such as ARM, ARM64 etc.

Debugging

Most of the internals of the IRQ subsystem are exposed in debugfs by turning CONFIG_GENERIC_IRQ_DEBUGFS on.

4.2.4 IRQ-flags state tracing

Author
started by Ingo Molnar <mingo@redhat.com>

The "irq-flags tracing" feature "traces" hardirq and softirq state, in that it gives interested subsystems an opportunity to be notified of every hardirqs-off/hardirqs-on, softirqs-off/softirqs-on event that happens in the kernel.

CONFIG_TRACE_IRQFLAGS_SUPPORT is needed for CONFIG_PROVE_SPIN_LOCKING and CONFIG_PROVE_RW_LOCKING to be offered by the generic lock debugging code. Otherwise only CONFIG_PROVE_MUTEX_LOCKING and CONFIG_PROVE_RWSEM_LOCKING will be offered on an architecture - these are locking APIs that are not used in IRQ context. (the one exception for rwsems is worked around)

Architecture support for this is certainly not in the "trivial" category, because lots of lowlevel assembly code deal with irq-flags state changes. But an architecture can be irq-flags-tracing enabled in a rather straightforward and risk-free manner.

Architectures that want to support this need to do a couple of code-organizational changes first:

- add and enable TRACE_IRQFLAGS_SUPPORT in their arch level Kconfig file

and then a couple of functional changes are needed as well to implement irq-flags-tracing support:

- in lowlevel entry code add (build-conditional) calls to the trace_hardirqs_off()/trace_hardirqs_on() functions. The lock validator closely guards whether the 'real' irq-flags matches the 'virtual' irq-flags state, and complains loudly (and turns itself off) if the two do not match. Usually most of the time for arch support for irq-flags-tracing is spent in this state: look at the lockdep complaint, try to figure out the assembly code we did not cover yet, fix and repeat. Once the system has booted up and works without a lockdep complaint in the irq-flags-tracing functions arch support is complete.
- if the architecture has non-maskable interrupts then those need to be excluded from the irq-tracing [and lock validation] mechanism via lockdep_off()/lockdep_on().

In general there is no risk from having an incomplete irq-flags-tracing implementation in an architecture: lockdep will detect that and will turn itself off. I.e. the lock validator will still be reliable. There should be no crashes due to irq-tracing bugs. (except if the assembly changes break other code by modifying conditions or registers that shouldn't be)

4.3 Semantics and Behavior of Local Atomic Operations

Author
Mathieu Desnoyers

This document explains the purpose of the local atomic operations, how to implement them for any given architecture and shows how they can be used properly. It also stresses on the precautions that must be taken when reading those local variables across CPUs when the order of memory writes matters.

Note: Note that `local_t` based operations are not recommended for general kernel use. Please use the `this_cpu` operations instead unless there is really a special purpose. Most uses of `local_t` in the kernel have been replaced by `this_cpu` operations. `this_cpu` operations combine the relocation with the `local_t` like semantics in a single instruction and yield more compact and faster executing code.

4.3.1 Purpose of local atomic operations

Local atomic operations are meant to provide fast and highly reentrant per CPU counters. They minimize the performance cost of standard atomic operations by removing the LOCK prefix and memory barriers normally required to synchronize across CPUs.

Having fast per CPU atomic counters is interesting in many cases: it does not require disabling interrupts to protect from interrupt handlers and it permits coherent counters in NMI handlers. It is especially useful for tracing purposes and for various performance monitoring counters.

Local atomic operations only guarantee variable modification atomicity wrt the CPU which owns the data. Therefore, care must taken to make sure that only one CPU writes to the `local_t` data. This is done by using per cpu data and making sure that we modify it from within a preemption safe context. It is however permitted to read `local_t` data from any CPU: it will then appear to be written out of order wrt other memory writes by the owner CPU.

4.3.2 Implementation for a given architecture

It can be done by slightly modifying the standard atomic operations: only their UP variant must be kept. It typically means removing LOCK prefix (on i386 and x86_64) and any SMP synchronization barrier. If the architecture does not have a different behavior between SMP and UP, including `asm-generic/local.h` in your architecture's `local.h` is sufficient.

The `local_t` type is defined as an opaque `signed long` by embedding an `atomic_long_t` inside a structure. This is made so a cast from this type to a `long` fails. The definition looks like:

```
typedef struct { atomic_long_t a; } local_t;
```

4.3.3 Rules to follow when using local atomic operations

- Variables touched by local ops must be per cpu variables.
- *Only* the CPU owner of these variables must write to them.
- This CPU can use local ops from any context (process, irq, softirq, nmi, …) to update its `local_t` variables.
- Preemption (or interrupts) must be disabled when using local ops in process context to make sure the process won't be migrated to a different CPU between getting the per-cpu variable and doing the actual local op.
- When using local ops in interrupt context, no special care must be taken on a mainline kernel, since they will run on the local CPU with preemption already disabled. I suggest, however, to explicitly disable preemption anyway to make sure it will still work correctly on -rt kernels.
- Reading the local cpu variable will provide the current copy of the variable.
- Reads of these variables can be done from any CPU, because updates to "`long`", aligned, variables are always atomic. Since no memory synchronization is done by the writer CPU, an outdated copy of the variable can be read when reading some *other* cpu's variables.

4.3.4 How to use local atomic operations

```
#include <linux/percpu.h>
#include <asm/local.h>

static DEFINE_PER_CPU(local_t, counters) = LOCAL_INIT(0);
```

4.3.5 Counting

Counting is done on all the bits of a signed long.

In preemptible context, use `get_cpu_var()` and `put_cpu_var()` around local atomic operations: it makes sure that preemption is disabled around write access to the per cpu variable. For instance:

```
local_inc(&get_cpu_var(counters));
put_cpu_var(counters);
```

If you are already in a preemption-safe context, you can use `this_cpu_ptr()` instead:

```
local_inc(this_cpu_ptr(&counters));
```

4.3.6 Reading the counters

Those local counters can be read from foreign CPUs to sum the count. Note that the data seen by local_read across CPUs must be considered to be out of order relatively to other memory writes happening on the CPU that owns the data:

```
long sum = 0;
for_each_online_cpu(cpu)
        sum += local_read(&per_cpu(counters, cpu));
```

If you want to use a remote local_read to synchronize access to a resource between CPUs, explicit `smp_wmb()` and `smp_rmb()` memory barriers must be used respectively on the writer and the reader CPUs. It would be the case if you use the `local_t` variable as a counter of bytes written in a buffer: there should be a `smp_wmb()` between the buffer write and the counter increment and also a `smp_rmb()` between the counter read and the buffer read.

Here is a sample module which implements a basic per cpu counter using `local.h`:

```
/* test-local.c
 *
 * Sample module for local.h usage.
 */


#include <asm/local.h>
#include <linux/module.h>
#include <linux/timer.h>

static DEFINE_PER_CPU(local_t, counters) = LOCAL_INIT(0);

static struct timer_list test_timer;

/* IPI called on each CPU. */
static void test_each(void *info)
{
        /* Increment the counter from a non preemptible context */
        printk("Increment on cpu %d\n", smp_processor_id());
        local_inc(this_cpu_ptr(&counters));

        /* This is what incrementing the variable would look like within a
         * preemptible context (it disables preemption) :
         *
         * local_inc(&get_cpu_var(counters));
         * put_cpu_var(counters);
         */
}

static void do_test_timer(unsigned long data)
{
        int cpu;

        /* Increment the counters */
```

```
        on_each_cpu(test_each, NULL, 1);
        /* Read all the counters */
        printk("Counters read from CPU %d\n", smp_processor_id());
        for_each_online_cpu(cpu) {
                printk("Read : CPU %d, count %ld\n", cpu,
                        local_read(&per_cpu(counters, cpu)));
        }
        mod_timer(&test_timer, jiffies + 1000);
}

static int __init test_init(void)
{
        /* initialize the timer that will increment the counter */
        timer_setup(&test_timer, do_test_timer, 0);
        mod_timer(&test_timer, jiffies + 1);

        return 0;
}

static void __exit test_exit(void)
{
        timer_shutdown_sync(&test_timer);
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mathieu Desnoyers");
MODULE_DESCRIPTION("Local Atomic Ops");
```

4.4 The padata parallel execution mechanism

Date
May 2020

Padata is a mechanism by which the kernel can farm jobs out to be done in parallel on multiple CPUs while optionally retaining their ordering.

It was originally developed for IPsec, which needs to perform encryption and decryption on large numbers of packets without reordering those packets. This is currently the sole consumer of padata's serialized job support.

Padata also supports multithreaded jobs, splitting up the job evenly while load balancing and coordinating between threads.

4.4.1 Running Serialized Jobs

Initializing

The first step in using padata to run serialized jobs is to set up a padata_instance structure for overall control of how jobs are to be run:

```
#include <linux/padata.h>

struct padata_instance *padata_alloc(const char *name);
```

'name' simply identifies the instance.

Then, complete padata initialization by allocating a padata_shell:

```
struct padata_shell *padata_alloc_shell(struct padata_instance *pinst);
```

A padata_shell is used to submit a job to padata and allows a series of such jobs to be serialized independently. A padata_instance may have one or more padata_shells associated with it, each allowing a separate series of jobs.

Modifying cpumasks

The CPUs used to run jobs can be changed in two ways, programmatically with padata_set_cpumask() or via sysfs. The former is defined:

```
int padata_set_cpumask(struct padata_instance *pinst, int cpumask_type,
                       cpumask_var_t cpumask);
```

Here cpumask_type is one of PADATA_CPU_PARALLEL or PADATA_CPU_SERIAL, where a parallel cpumask describes which processors will be used to execute jobs submitted to this instance in parallel and a serial cpumask defines which processors are allowed to be used as the serialization callback processor. cpumask specifies the new cpumask to use.

There may be sysfs files for an instance's cpumasks. For example, pcrypt's live in /sys/kernel/pcrypt/<instance-name>. Within an instance's directory there are two files, parallel_cpumask and serial_cpumask, and either cpumask may be changed by echoing a bitmask into the file, for example:

```
echo f > /sys/kernel/pcrypt/pencrypt/parallel_cpumask
```

Reading one of these files shows the user-supplied cpumask, which may be different from the 'usable' cpumask.

Padata maintains two pairs of cpumasks internally, the user-supplied cpumasks and the 'usable' cpumasks. (Each pair consists of a parallel and a serial cpumask.) The user-supplied cpumasks default to all possible CPUs on instance allocation and may be changed as above. The usable cpumasks are always a subset of the user-supplied cpumasks and contain only the online CPUs in the user-supplied masks; these are the cpumasks padata actually uses. So it is legal to supply a cpumask to padata that contains offline CPUs. Once an offline CPU in the user-supplied cpumask comes online, padata is going to use it.

Changing the CPU masks are expensive operations, so it should not be done with great frequency.

Running A Job

Actually submitting work to the padata instance requires the creation of a padata_priv structure, which represents one job:

```
struct padata_priv {
    /* Other stuff here... */
    void                    (*parallel)(struct padata_priv *padata);
    void                    (*serial)(struct padata_priv *padata);
};
```

This structure will almost certainly be embedded within some larger structure specific to the work to be done. Most of its fields are private to padata, but the structure should be zeroed at initialisation time, and the parallel() and serial() functions should be provided. Those functions will be called in the process of getting the work done as we will see momentarily.

The submission of the job is done with:

```
int padata_do_parallel(struct padata_shell *ps,
                       struct padata_priv *padata, int *cb_cpu);
```

The ps and padata structures must be set up as described above; cb_cpu points to the preferred CPU to be used for the final callback when the job is done; it must be in the current instance's CPU mask (if not the cb_cpu pointer is updated to point to the CPU actually chosen). The return value from padata_do_parallel() is zero on success, indicating that the job is in progress. -EBUSY means that somebody, somewhere else is messing with the instance's CPU mask, while -EINVAL is a complaint about cb_cpu not being in the serial cpumask, no online CPUs in the parallel or serial cpumasks, or a stopped instance.

Each job submitted to padata_do_parallel() will, in turn, be passed to exactly one call to the above-mentioned parallel() function, on one CPU, so true parallelism is achieved by submitting multiple jobs. parallel() runs with software interrupts disabled and thus cannot sleep. The parallel() function gets the padata_priv structure pointer as its lone parameter; information about the actual work to be done is probably obtained by using container_of() to find the enclosing structure.

Note that parallel() has no return value; the padata subsystem assumes that parallel() will take responsibility for the job from this point. The job need not be completed during this call, but, if parallel() leaves work outstanding, it should be prepared to be called again with a new job before the previous one completes.

Serializing Jobs

When a job does complete, parallel() (or whatever function actually finishes the work) should inform padata of the fact with a call to:

```
void padata_do_serial(struct padata_priv *padata);
```

At some point in the future, padata_do_serial() will trigger a call to the serial() function in the padata_priv structure. That call will happen on the CPU requested in the initial call to padata_do_parallel(); it, too, is run with local software interrupts disabled. Note that this call may be deferred for a while since the padata code takes pains to ensure that jobs are completed in the order in which they were submitted.

Destroying

Cleaning up a padata instance predictably involves calling the two free functions that correspond to the allocation in reverse:

```
void padata_free_shell(struct padata_shell *ps);
void padata_free(struct padata_instance *pinst);
```

It is the user's responsibility to ensure all outstanding jobs are complete before any of the above are called.

4.4.2 Running Multithreaded Jobs

A multithreaded job has a main thread and zero or more helper threads, with the main thread participating in the job and then waiting until all helpers have finished. padata splits the job into units called chunks, where a chunk is a piece of the job that one thread completes in one call to the thread function.

A user has to do three things to run a multithreaded job. First, describe the job by defining a padata_mt_job structure, which is explained in the Interface section. This includes a pointer to the thread function, which padata will call each time it assigns a job chunk to a thread. Then, define the thread function, which accepts three arguments, `start`, `end`, and `arg`, where the first two delimit the range that the thread operates on and the last is a pointer to the job's shared state, if any. Prepare the shared state, which is typically allocated on the main thread's stack. Last, call padata_do_multithreaded(), which will return once the job is finished.

4.4.3 Interface

struct **padata_priv**

Represents one job

Definition:

```
struct padata_priv {
    struct list_head        list;
    struct parallel_data    *pd;
    int cb_cpu;
    unsigned int            seq_nr;
    int info;
    void (*parallel)(struct padata_priv *padata);
    void (*serial)(struct padata_priv *padata);
};
```

Members

list
List entry, to attach to the padata lists.

pd
Pointer to the internal control structure.

cb_cpu
Callback cpu for serializatioon.

seq_nr
Sequence number of the parallelized data object.

info
Used to pass information from the parallel to the serial function.

parallel
Parallel execution function.

serial
Serial complete function.

struct **padata_list**
one per work type per CPU

Definition:

```
struct padata_list {
    struct list_head        list;
    spinlock_t lock;
};
```

Members

list
List head.

lock
List lock.

struct **padata_serial_queue**
The percpu padata serial queue

Definition:

```
struct padata_serial_queue {
    struct padata_list    serial;
    struct work_struct    work;
    struct parallel_data *pd;
};
```

Members

serial
List to wait for serialization after reordering.

work
work struct for serialization.

pd
Backpointer to the internal control structure.

struct **padata_cpumask**
The cpumasks for the parallel/serial workers

Definition:

```
struct padata_cpumask {
    cpumask_var_t pcpu;
    cpumask_var_t cbcpu;
};
```

Members

pcpu
cpumask for the parallel workers.

cbcpu
cpumask for the serial (callback) workers.

struct **parallel_data**
Internal control structure, covers everything that depends on the cpumask in use.

Definition:

```
struct parallel_data {
    struct padata_shell             *ps;
    struct padata_list              __percpu *reorder_list;
    struct padata_serial_queue      __percpu *squeue;
    refcount_t refcnt;
    unsigned int                    seq_nr;
    unsigned int                    processed;
    int cpu;
    struct padata_cpumask           cpumask;
    struct work_struct              reorder_work;
    spinlock_t lock;
};
```

Members

ps
padata_shell object.

reorder_list
percpu reorder lists

squeue
percpu padata queues used for serialuzation.

refcnt
Number of objects holding a reference on this parallel_data.

seq_nr
Sequence number of the parallelized data object.

processed
Number of already processed objects.

cpu
Next CPU to be processed.

cpumask
The cpumasks in use for parallel and serial workers.

reorder_work
work struct for reordering.

lock
Reorder lock.

struct **padata_shell**
Wrapper around *struct parallel_data*, its purpose is to allow the underlying control structure to be replaced on the fly using RCU.

Definition:

```
struct padata_shell {
    struct padata_instance          *pinst;
    struct parallel_data __rcu      *pd;
    struct parallel_data            *opd;
    struct list_head                list;
};
```

Members

pinst
padat instance.

pd
Actual parallel_data structure which may be substituted on the fly.

opd
Pointer to old pd to be freed by padata_replace.

list
List entry in padata_instance list.

struct **padata_mt_job**
represents one multithreaded job

Definition:

```
struct padata_mt_job {
    void (*thread_fn)(unsigned long start, unsigned long end, void *arg);
    void *fn_arg;
    unsigned long           start;
    unsigned long           size;
    unsigned long           align;
    unsigned long           min_chunk;
    int max_threads;
};
```

Members

thread_fn
Called for each chunk of work that a padata thread does.

fn_arg
The thread function argument.

start
The start of the job (units are job-specific).

size
size of this node's work (units are job-specific).

align
Ranges passed to the thread function fall on this boundary, with the possible exceptions of the beginning and end of the job.

min_chunk
The minimum chunk size in job-specific units. This allows the client to communicate the minimum amount of work that's appropriate for one worker thread to do at once.

max_threads
Max threads to use for the job, actual number may be less depending on task size and minimum chunk size.

struct **padata_instance**

The overall control structure.

Definition:

```
struct padata_instance {
    struct hlist_node               cpu_online_node;
    struct hlist_node               cpu_dead_node;
    struct workqueue_struct         *parallel_wq;
    struct workqueue_struct         *serial_wq;
    struct list_head                pslist;
    struct padata_cpumask           cpumask;
    struct kobject                   kobj;
    struct mutex                     lock;
    u8 flags;
#define PADATA_INIT     1;
#define PADATA_RESET    2;
#define PADATA_INVALID  4;
};
```

Members

cpu_online_node
Linkage for CPU online callback.

cpu_dead_node
Linkage for CPU offline callback.

parallel_wq
The workqueue used for parallel work.

serial_wq
The workqueue used for serial work.

pslist
List of padata_shell objects attached to this instance.

cpumask
User supplied cpumasks for parallel and serial works.

kobj
padata instance kernel object.

lock
padata instance lock.

flags
padata flags.

int **padata_do_parallel**(struct *padata_shell* *ps, struct *padata_priv* *padata, int *cb_cpu)
padata parallelization function

Parameters

struct padata_shell *ps
padatashell

struct padata_priv *padata
object to be parallelized

int *cb_cpu
pointer to the CPU that the serialization callback function should run on. If it's not in the serial cpumask of **pinst** (i.e. cpumask.cbcpu), this function selects a fallback CPU and if none found, returns -EINVAL.

Description

The parallelization callback function will run with BHs off.

Note

Every object which is parallelized by padata_do_parallel must be seen by padata_do_serial.

Return

0 on success or else negative error code.

void **padata_do_serial**(struct *padata_priv* *padata)
padata serialization function

Parameters

struct padata_priv *padata
object to be serialized.

Description

padata_do_serial must be called for every parallelized object. The serialization callback function will run with BHs off.

void **padata_do_multithreaded**(struct *padata_mt_job* *job)
run a multithreaded job

Parameters

struct padata_mt_job *job
Description of the job.

Description

See the definition of *struct padata_mt_job* for more details.

int **padata_set_cpumask**(struct *padata_instance* *pinst, int cpumask_type, cpumask_var_t cpumask)

Sets specified by **cpumask_type** cpumask to the value equivalent to **cpumask**.

Parameters

struct padata_instance *pinst
padata instance

int cpumask_type
PADATA_CPU_SERIAL or PADATA_CPU_PARALLEL corresponding to parallel and serial cpumasks respectively.

cpumask_var_t cpumask
the cpumask to use

Return

0 on success or negative error code

struct *padata_instance* ***padata_alloc**(const char *name)

allocate and initialize a padata instance

Parameters

const char *name
used to identify the instance

Return

new instance on success, NULL on error

void **padata_free**(struct *padata_instance* *pinst)

free a padata instance

Parameters

struct padata_instance *pinst
padata instance to free

struct *padata_shell* ***padata_alloc_shell**(struct *padata_instance* *pinst)

Allocate and initialize padata shell.

Parameters

struct padata_instance *pinst
Parent padata_instance object.

Return

new shell on success, NULL on error

void **padata_free_shell**(struct *padata_shell* *ps)

free a padata shell

Parameters

struct padata_shell *ps
padata shell to free

4.5 RCU concepts

4.5.1 Review Checklist for RCU Patches

This document contains a checklist for producing and reviewing patches that make use of RCU. Violating any of the rules listed below will result in the same sorts of problems that leaving out a locking primitive would cause. This list is based on experiences reviewing such patches over a rather long period of time, but improvements are always welcome!

0. Is RCU being applied to a read-mostly situation? If the data structure is updated more than about 10% of the time, then you should strongly consider some other approach, unless detailed performance measurements show that RCU is nonetheless the right tool for the job. Yes, RCU does reduce read-side overhead by increasing write-side overhead, which is exactly why normal uses of RCU will do much more reading than updating.

   Another exception is where performance is not an issue, and RCU provides a simpler implementation. An example of this situation is the dynamic NMI code in the Linux 2.6 kernel, at least on architectures where NMIs are rare.

   Yet another exception is where the low real-time latency of RCU's read-side primitives is critically important.

   One final exception is where RCU readers are used to prevent the ABA problem (https://en.wikipedia.org/wiki/ABA_problem) for lockless updates. This does result in the mildly counter-intuitive situation where rcu_read_lock() and rcu_read_unlock() are used to protect updates, however, this approach can provide the same simplifications to certain types of lockless algorithms that garbage collectors do.

1. Does the update code have proper mutual exclusion?

   RCU does allow *readers* to run (almost) naked, but *writers* must still use some sort of mutual exclusion, such as:

   a. locking,

   b. atomic operations, or

   c. restricting updates to a single task.

   If you choose #b, be prepared to describe how you have handled memory barriers on weakly ordered machines (pretty much all of them -- even x86 allows later loads to be reordered to precede earlier stores), and be prepared to explain why this added complexity is worthwhile. If you choose #c, be prepared to explain how this single task does not become a major bottleneck on large systems (for example, if the task is updating information relating to itself that other tasks can read, there by definition can be no bottleneck). Note that the definition of "large" has changed significantly: Eight CPUs was "large" in the year 2000, but a hundred CPUs was unremarkable in 2017.

2. Do the RCU read-side critical sections make proper use of rcu_read_lock() and friends? These primitives are needed to prevent grace periods from ending prematurely, which could result in data being unceremoniously freed out from under your read-side code, which can greatly increase the actuarial risk of your kernel.

   As a rough rule of thumb, any dereference of an RCU-protected pointer must be covered by rcu_read_lock(), `rcu_read_lock_bh()`, `rcu_read_lock_sched()`, or by the appropriate update-side lock. Explicit disabling of preemption (preempt_disable(), for example) can serve as `rcu_read_lock_sched()`, but is less readable and prevents lockdep from detecting locking issues.

Please note that you *cannot* rely on code known to be built only in non-preemptible kernels. Such code can and will break, especially in kernels built with CONFIG_PREEMPT_COUNT=y.

Letting RCU-protected pointers "leak" out of an RCU read-side critical section is every bit as bad as letting them leak out from under a lock. Unless, of course, you have arranged some other means of protection, such as a lock or a reference count *before* letting them out of the RCU read-side critical section.

3. Does the update code tolerate concurrent accesses?

   The whole point of RCU is to permit readers to run without any locks or atomic operations. This means that readers will be running while updates are in progress. There are a number of ways to handle this concurrency, depending on the situation:

   a. Use the RCU variants of the list and hlist update primitives to add, remove, and replace elements on an RCU-protected list. Alternatively, use the other RCU-protected data structures that have been added to the Linux kernel.

      This is almost always the best approach.

   b. Proceed as in (a) above, but also maintain per-element locks (that are acquired by both readers and writers) that guard per-element state. Fields that the readers refrain from accessing can be guarded by some other lock acquired only by updaters, if desired.

      This also works quite well.

   c. Make updates appear atomic to readers. For example, pointer updates to properly aligned fields will appear atomic, as will individual atomic primitives. Sequences of operations performed under a lock will *not* appear to be atomic to RCU readers, nor will sequences of multiple atomic primitives. One alternative is to move multiple individual fields to a separate structure, thus solving the multiple-field problem by imposing an additional level of indirection.

      This can work, but is starting to get a bit tricky.

   d. Carefully order the updates and the reads so that readers see valid data at all phases of the update. This is often more difficult than it sounds, especially given modern CPUs' tendency to reorder memory references. One must usually liberally sprinkle memory-ordering operations through the code, making it difficult to understand and to test. Where it works, it is better to use things like smp_store_release() and smp_load_acquire(), but in some cases the smp_mb() full memory barrier is required.

      As noted earlier, it is usually better to group the changing data into a separate structure, so that the change may be made to appear atomic by updating a pointer to reference a new structure containing updated values.

4. Weakly ordered CPUs pose special challenges. Almost all CPUs are weakly ordered -- even x86 CPUs allow later loads to be reordered to precede earlier stores. RCU code must take all of the following measures to prevent memory-corruption problems:

   a. Readers must maintain proper ordering of their memory accesses. The rcu_dereference() primitive ensures that the CPU picks up the pointer before it picks up the data that the pointer points to. This really is necessary on Alpha CPUs.

The rcu_dereference() primitive is also an excellent documentation aid, letting the person reading the code know exactly which pointers are protected by RCU. Please note that compilers can also reorder code, and they are becoming increasingly aggressive about doing just that. The rcu_dereference() primitive therefore also prevents destructive compiler optimizations. However, with a bit of devious creativity, it is possible to mishandle the return value from rcu_dereference(). Please see rcu_dereference.rst for more information.

The rcu_dereference() primitive is used by the various "_rcu()" list-traversal primitives, such as the list_for_each_entry_rcu(). Note that it is perfectly legal (if redundant) for update-side code to use rcu_dereference() and the "_rcu()" list-traversal primitives. This is particularly useful in code that is common to readers and updaters. However, lockdep will complain if you access rcu_dereference() outside of an RCU read-side critical section. See lockdep.rst to learn what to do about this.

Of course, neither rcu_dereference() nor the "_rcu()" list-traversal primitives can substitute for a good concurrency design coordinating among multiple updaters.

b. If the list macros are being used, the `list_add_tail_rcu()` and list_add_rcu() primitives must be used in order to prevent weakly ordered machines from misordering structure initialization and pointer planting. Similarly, if the hlist macros are being used, the hlist_add_head_rcu() primitive is required.

c. If the list macros are being used, the list_del_rcu() primitive must be used to keep `list_del()`'s pointer poisoning from inflicting toxic effects on concurrent readers. Similarly, if the hlist macros are being used, the `hlist_del_rcu()` primitive is required.

The `list_replace_rcu()` and `hlist_replace_rcu()` primitives may be used to replace an old structure with a new one in their respective types of RCU-protected lists.

d. Rules similar to (4b) and (4c) apply to the "hlist_nulls" type of RCU-protected linked lists.

e. Updates must ensure that initialization of a given structure happens before pointers to that structure are publicized. Use the rcu_assign_pointer() primitive when publicizing a pointer to a structure that can be traversed by an RCU read-side critical section.

5. If any of call_rcu(), `call_srcu()`, `call_rcu_tasks()`, `call_rcu_tasks_rude()`, or `call_rcu_tasks_trace()` is used, the callback function may be invoked from softirq context, and in any case with bottom halves disabled. In particular, this callback function cannot block. If you need the callback to block, run that code in a workqueue handler scheduled from the callback. The `queue_rcu_work()` function does this for you in the case of call_rcu().

6. Since synchronize_rcu() can block, it cannot be called from any sort of irq context. The same rule applies for synchronize_srcu(), `synchronize_rcu_expedited()`, `synchronize_srcu_expedited()`, `synchronize_rcu_tasks()`, `synchronize_rcu_tasks_rude()`, and `synchronize_rcu_tasks_trace()`.

   The expedited forms of these primitives have the same semantics as the non-expedited forms, but expediting is more CPU intensive. Use of the expedited primitives should be restricted to rare configuration-change operations that would not normally be undertaken while a real-time workload is running. Note that IPI-sensitive real-time workloads can use the rcupdate.rcu_normal kernel boot parameter to completely disable expedited grace periods, though this might have performance implications.

In particular, if you find yourself invoking one of the expedited primitives repeatedly in a loop, please do everyone a favor: Restructure your code so that it batches the updates, allowing a single non-expedited primitive to cover the entire batch. This will very likely be faster than the loop containing the expedited primitive, and will be much much easier on the rest of the system, especially to real-time workloads running on the rest of the system. Alternatively, instead use asynchronous primitives such as call_rcu().

7. As of v4.20, a given kernel implements only one RCU flavor, which is RCU-sched for PREEMPTION=n and RCU-preempt for PREEMPTION=y. If the updater uses call_rcu() or synchronize_rcu(), then the corresponding readers may use: (1) rcu_read_lock() and rcu_read_unlock(), (2) any pair of primitives that disables and re-enables softirq, for example, `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`, or (3) any pair of primitives that disables and re-enables preemption, for example, `rcu_read_lock_sched()` and `rcu_read_unlock_sched()`. If the updater uses synchronize_srcu() or `call_srcu()`, then the corresponding readers must use `srcu_read_lock()` and `srcu_read_unlock()`, and with the same srcu_struct. The rules for the expedited RCU grace-period-wait primitives are the same as for their non-expedited counterparts.

   If the updater uses `call_rcu_tasks()` or `synchronize_rcu_tasks()`, then the readers must refrain from executing voluntary context switches, that is, from blocking. If the updater uses `call_rcu_tasks_trace()` or `synchronize_rcu_tasks_trace()`, then the corresponding readers must use `rcu_read_lock_trace()` and `rcu_read_unlock_trace()`. If an updater uses `call_rcu_tasks_rude()` or `synchronize_rcu_tasks_rude()`, then the corresponding readers must use anything that disables preemption, for example, preempt_disable() and preempt_enable().

   Mixing things up will result in confusion and broken kernels, and has even resulted in an exploitable security issue. Therefore, when using non-obvious pairs of primitives, commenting is of course a must. One example of non-obvious pairing is the XDP feature in networking, which calls BPF programs from network-driver NAPI (softirq) context. BPF relies heavily on RCU protection for its data structures, but because the BPF program invocation happens entirely within a single local_bh_disable() section in a NAPI poll cycle, this usage is safe. The reason that this usage is safe is that readers can use anything that disables BH when updaters use call_rcu() or synchronize_rcu().

8. Although synchronize_rcu() is slower than is call_rcu(), it usually results in simpler code. So, unless update performance is critically important, the updaters cannot block, or the latency of synchronize_rcu() is visible from userspace, synchronize_rcu() should be used in preference to call_rcu(). Furthermore, kfree_rcu() and kvfree_rcu() usually result in even simpler code than does synchronize_rcu() without synchronize_rcu()'s multi-millisecond latency. So please take advantage of kfree_rcu()'s and kvfree_rcu()'s "fire and forget" memory-freeing capabilities where it applies.

   An especially important property of the synchronize_rcu() primitive is that it automatically self-limits: if grace periods are delayed for whatever reason, then the synchronize_rcu() primitive will correspondingly delay updates. In contrast, code using call_rcu() should explicitly limit update rate in cases where grace periods are delayed, as failing to do so can result in excessive realtime latencies or even OOM conditions.

   Ways of gaining this self-limiting property when using call_rcu(), kfree_rcu(), or kvfree_rcu() include:

a. Keeping a count of the number of data-structure elements used by the RCU-protected data structure, including those waiting for a grace period to elapse. Enforce a limit on this number, stalling updates as needed to allow previously deferred frees to complete. Alternatively, limit only the number awaiting deferred free rather than the total number of elements.

   One way to stall the updates is to acquire the update-side mutex. (Don't try this with a spinlock -- other CPUs spinning on the lock could prevent the grace period from ever ending.) Another way to stall the updates is for the updates to use a wrapper function around the memory allocator, so that this wrapper function simulates OOM when there is too much memory awaiting an RCU grace period. There are of course many other variations on this theme.

b. Limiting update rate. For example, if updates occur only once per hour, then no explicit rate limiting is required, unless your system is already badly broken. Older versions of the dcache subsystem take this approach, guarding updates with a global lock, limiting their rate.

c. Trusted update -- if updates can only be done manually by superuser or some other trusted user, then it might not be necessary to automatically limit them. The theory here is that superuser already has lots of ways to crash the machine.

d. Periodically invoke `rcu_barrier()`, permitting a limited number of updates per grace period.

The same cautions apply to `call_srcu()`, `call_rcu_tasks()`, `call_rcu_tasks_rude()`, and `call_rcu_tasks_trace()`. This is why there is an `srcu_barrier()`, `rcu_barrier_tasks()`, `rcu_barrier_tasks_rude()`, and `rcu_barrier_tasks_rude()`, respectively.

Note that although these primitives do take action to avoid memory exhaustion when any given CPU has too many callbacks, a determined user or administrator can still exhaust memory. This is especially the case if a system with a large number of CPUs has been configured to offload all of its RCU callbacks onto a single CPU, or if the system has relatively little free memory.

9. All RCU list-traversal primitives, which include rcu_dereference(), list_for_each_entry_rcu(), and list_for_each_safe_rcu(), must be either within an RCU read-side critical section or must be protected by appropriate update-side locks. RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), or by similar primitives such as `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`, in which case the matching rcu_dereference() primitive must be used in order to keep lockdep happy, in this case, `rcu_dereference_bh()`.

   The reason that it is permissible to use RCU list-traversal primitives when the update-side lock is held is that doing so can be quite helpful in reducing code bloat when common code is shared between readers and updaters. Additional primitives are provided for this case, as discussed in lockdep.rst.

   One exception to this rule is when data is only ever added to the linked data structure, and is never removed during any time that readers might be accessing that structure. In such cases, READ_ONCE() may be used in place of rcu_dereference() and the read-side markers (rcu_read_lock() and rcu_read_unlock(), for example) may be omitted.

10. Conversely, if you are in an RCU read-side critical section, and you don't hold the appropriate update-side lock, you *must* use the "_rcu()" variants of the list macros. Failing to do

so will break Alpha, cause aggressive compilers to generate bad code, and confuse people trying to understand your code.

11. Any lock acquired by an RCU callback must be acquired elsewhere with softirq disabled, e.g., via spin_lock_bh(). Failing to disable softirq on a given acquisition of that lock will result in deadlock as soon as the RCU softirq handler happens to run your RCU callback while interrupting that acquisition's critical section.

12. RCU callbacks can be and are executed in parallel. In many cases, the callback code simply wrappers around kfree(), so that this is not an issue (or, more accurately, to the extent that it is an issue, the memory-allocator locking handles it). However, if the callbacks do manipulate a shared data structure, they must use whatever locking or other synchronization is required to safely access and/or modify that data structure.

    Do not assume that RCU callbacks will be executed on the same CPU that executed the corresponding call_rcu() or `call_srcu()`. For example, if a given CPU goes offline while having an RCU callback pending, then that RCU callback will execute on some surviving CPU. (If this was not the case, a self-spawning RCU callback would prevent the victim CPU from ever going offline.) Furthermore, CPUs designated by rcu_nocbs= might well *always* have their RCU callbacks executed on some other CPUs, in fact, for some real-time workloads, this is the whole point of using the rcu_nocbs= kernel boot parameter.

    In addition, do not assume that callbacks queued in a given order will be invoked in that order, even if they all are queued on the same CPU. Furthermore, do not assume that same-CPU callbacks will be invoked serially. For example, in recent kernels, CPUs can be switched between offloaded and de-offloaded callback invocation, and while a given CPU is undergoing such a switch, its callbacks might be concurrently invoked by that CPU's softirq handler and that CPU's rcuo kthread. At such times, that CPU's callbacks might be executed both concurrently and out of order.

13. Unlike most flavors of RCU, it *is* permissible to block in an SRCU read-side critical section (demarked by `srcu_read_lock()` and `srcu_read_unlock()`), hence the "SRCU": "sleepable RCU". Please note that if you don't need to sleep in read-side critical sections, you should be using RCU rather than SRCU, because RCU is almost always faster and easier to use than is SRCU.

    Also unlike other forms of RCU, explicit initialization and cleanup is required either at build time via DEFINE_SRCU() or DEFINE_STATIC_SRCU() or at runtime via `init_srcu_struct()` and `cleanup_srcu_struct()`. These last two are passed a "struct srcu_struct" that defines the scope of a given SRCU domain. Once initialized, the srcu_struct is passed to `srcu_read_lock()`, `srcu_read_unlock()` synchronize_srcu(), `synchronize_srcu_expedited()`, and `call_srcu()`. A given synchronize_srcu() waits only for SRCU read-side critical sections governed by `srcu_read_lock()` and `srcu_read_unlock()` calls that have been passed the same srcu_struct. This property is what makes sleeping read-side critical sections tolerable -- a given subsystem delays only its own updates, not those of other subsystems using SRCU. Therefore, SRCU is less prone to OOM the system than RCU would be if RCU's read-side critical sections were permitted to sleep.

    The ability to sleep in read-side critical sections does not come for free. First, corresponding `srcu_read_lock()` and `srcu_read_unlock()` calls must be passed the same srcu_struct. Second, grace-period-detection overhead is amortized only over those updates sharing a given srcu_struct, rather than being globally amortized as they are for other forms of RCU. Therefore, SRCU should be used in preference to rw_semaphore only in extremely read-intensive situations, or in situations requiring SRCU's read-side deadlock immunity or low read-side realtime latency. You should also consider percpu_rw_semaphore when you need lightweight readers.

SRCU's expedited primitive (*synchronize_srcu_expedited()*) never sends IPIs to other CPUs, so it is easier on real-time workloads than is *synchronize_rcu_expedited()*.

It is also permissible to sleep in RCU Tasks Trace read-side critical, which are delimited by *rcu_read_lock_trace()* and *rcu_read_unlock_trace()*. However, this is a specialized flavor of RCU, and you should not use it without first checking with its current users. In most cases, you should instead use SRCU.

Note that rcu_assign_pointer() relates to SRCU just as it does to other forms of RCU, but instead of rcu_dereference() you should use *srcu_dereference()* in order to avoid lockdep splats.

14. The whole point of call_rcu(), synchronize_rcu(), and friends is to wait until all pre-existing readers have finished before carrying out some otherwise-destructive operation. It is therefore critically important to *first* remove any path that readers can follow that could be affected by the destructive operation, and *only then* invoke call_rcu(), synchronize_rcu(), or friends.

    Because these primitives only wait for pre-existing readers, it is the caller's responsibility to guarantee that any subsequent readers will execute safely.

15. The various RCU read-side primitives do *not* necessarily contain memory barriers. You should therefore plan for the CPU and the compiler to freely reorder code into and out of RCU read-side critical sections. It is the responsibility of the RCU update-side primitives to deal with this.

    For SRCU readers, you can use *smp_mb__after_srcu_read_unlock()* immediately after an *srcu_read_unlock()* to get a full barrier.

16. Use CONFIG_PROVE_LOCKING, CONFIG_DEBUG_OBJECTS_RCU_HEAD, and the __rcu sparse checks to validate your RCU code. These can help find problems as follows:

    CONFIG_PROVE_LOCKING:
    check that accesses to RCU-protected data structures are carried out under the proper RCU read-side critical section, while holding the right combination of locks, or whatever other conditions are appropriate.

    CONFIG_DEBUG_OBJECTS_RCU_HEAD:
    check that you don't pass the same object to call_rcu() (or friends) before an RCU grace period has elapsed since the last time that you passed that same object to call_rcu() (or friends).

    __rcu sparse checks:
    tag the pointer to the RCU-protected data structure with __rcu, and sparse will warn you if you access that pointer without the services of one of the variants of rcu_dereference().

    These debugging aids can help you find problems that are otherwise extremely difficult to spot.

17. If you pass a callback function defined within a module to one of call_rcu(), *call_srcu()*, *call_rcu_tasks()*, *call_rcu_tasks_rude()*, or *call_rcu_tasks_trace()*, then it is necessary to wait for all pending callbacks to be invoked before unloading that module. Note that it is absolutely *not* sufficient to wait for a grace period! For example, synchronize_rcu() implementation is *not* guaranteed to wait for callbacks registered on other CPUs

via call_rcu(). Or even on the current CPU if that CPU recently went offline and came back online.

You instead need to use one of the barrier functions:

- call_rcu() -> *rcu_barrier()*
- *call_srcu()* -> *srcu_barrier()*
- *call_rcu_tasks()* -> *rcu_barrier_tasks()*
- *call_rcu_tasks_rude()* -> *rcu_barrier_tasks_rude()*
- *call_rcu_tasks_trace()* -> *rcu_barrier_tasks_trace()*

However, these barrier functions are absolutely *not* guaranteed to wait for a grace period. For example, if there are no call_rcu() callbacks queued anywhere in the system, *rcu_barrier()* can and will return immediately.

So if you need to wait for both a grace period and for all pre-existing callbacks, you will need to invoke both functions, with the pair depending on the flavor of RCU:

- Either synchronize_rcu() or *synchronize_rcu_expedited()*, together with *rcu_barrier()*
- Either synchronize_srcu() or *synchronize_srcu_expedited()*, together with and *srcu_barrier()*
- *synchronize_rcu_tasks()* and *rcu_barrier_tasks()*
- synchronize_tasks_rude() and *rcu_barrier_tasks_rude()*
- synchronize_tasks_trace() and *rcu_barrier_tasks_trace()*

If necessary, you can use something like workqueues to execute the requisite pair of functions concurrently.

See rcubarrier.rst for more information.

4.5.2 RCU and lockdep checking

All flavors of RCU have lockdep checking available, so that lockdep is aware of when each task enters and leaves any flavor of RCU read-side critical section. Each flavor of RCU is tracked separately (but note that this is not the case in 2.6.32 and earlier). This allows lockdep's tracking to include RCU state, which can sometimes help when debugging deadlocks and the like.

In addition, RCU provides the following primitives that check lockdep's state:

```
rcu_read_lock_held() for normal RCU.
rcu_read_lock_bh_held() for RCU-bh.
rcu_read_lock_sched_held() for RCU-sched.
rcu_read_lock_any_held() for any of normal RCU, RCU-bh, and RCU-sched.
srcu_read_lock_held() for SRCU.
rcu_read_lock_trace_held() for RCU Tasks Trace.
```

These functions are conservative, and will therefore return 1 if they aren't certain (for example, if CONFIG_DEBUG_LOCK_ALLOC is not set). This prevents things like WARN_ON(!*rcu_read_lock_held()*) from giving false positives when lockdep is disabled.

In addition, a separate kernel config parameter CONFIG_PROVE_RCU enables checking of rcu_dereference() primitives:

rcu_dereference(p):
Check for RCU read-side critical section.

rcu_dereference_bh(p):
Check for RCU-bh read-side critical section.

rcu_dereference_sched(p):
Check for RCU-sched read-side critical section.

srcu_dereference(p, sp):
Check for SRCU read-side critical section.

rcu_dereference_check(p, c):
Use explicit check expression "c" along with *rcu_read_lock_held()*. This is useful in code that is invoked by both RCU readers and updaters.

rcu_dereference_bh_check(p, c):
Use explicit check expression "c" along with *rcu_read_lock_bh_held()*. This is useful in code that is invoked by both RCU-bh readers and updaters.

rcu_dereference_sched_check(p, c):
Use explicit check expression "c" along with rcu_read_lock_sched_held(). This is useful in code that is invoked by both RCU-sched readers and updaters.

srcu_dereference_check(p, c):
Use explicit check expression "c" along with *srcu_read_lock_held()*. This is useful in code that is invoked by both SRCU readers and updaters.

rcu_dereference_raw(p):
Don't check. (Use sparingly, if at all.)

rcu_dereference_raw_check(p):
Don't do lockdep at all. (Use sparingly, if at all.)

rcu_dereference_protected(p, c):
Use explicit check expression "c", and omit all barriers and compiler constraints. This is useful when the data structure cannot change, for example, in code that is invoked only by updaters.

rcu_access_pointer(p):
Return the value of the pointer and omit all barriers, but retain the compiler constraints that prevent duplicating or coalescing. This is useful when testing the value of the pointer itself, for example, against NULL.

The *rcu_dereference_check()* check expression can be any boolean expression, but would normally include a lockdep expression. For a moderately ornate example, consider the following:

```
file = rcu_dereference_check(fdt->fd[fd],
                             lockdep_is_held(&files->file_lock) ||
                             atomic_read(&files->count) == 1);
```

This expression picks up the pointer "fdt->fd[fd]" in an RCU-safe manner, and, if CONFIG_PROVE_RCU is configured, verifies that this expression is used in:

1. An RCU read-side critical section (implicit), or
2. with files->file_lock held, or
3. on an unshared files_struct.

In case (1), the pointer is picked up in an RCU-safe manner for vanilla RCU read-side critical sections, in case (2) the ->file_lock prevents any change from taking place, and finally, in case (3) the current task is the only task accessing the file_struct, again preventing any change from taking place. If the above statement was invoked only from updater code, it could instead be written as follows:

```
file = rcu_dereference_protected(fdt->fd[fd],
                                 lockdep_is_held(&files->file_lock) ||
                                 atomic_read(&files->count) == 1);
```

This would verify cases #2 and #3 above, and furthermore lockdep would complain even if this was used in an RCU read-side critical section unless one of these two cases held. Because *rcu_dereference_protected()* omits all barriers and compiler constraints, it generates better code than do the other flavors of rcu_dereference(). On the other hand, it is illegal to use *rcu_dereference_protected()* if either the RCU-protected pointer or the RCU-protected data that it points to can change concurrently.

Like rcu_dereference(), when lockdep is enabled, RCU list and hlist traversal primitives check for being called from within an RCU read-side critical section. However, a lockdep expression can be passed to them as a additional optional argument. With this lockdep expression, these traversal primitives will complain only if the lockdep expression is false and they are called from outside any RCU read-side critical section.

For example, the workqueue *for_each_pwq()* macro is intended to be used either within an RCU read-side critical section or with wq->mutex held. It is thus implemented as follows:

```
#define for_each_pwq(pwq, wq)
        list_for_each_entry_rcu((pwq), &(wq)->pwqs, pwqs_node,
                                lock_is_held(&(wq->mutex).dep_map))
```

4.5.3 Lockdep-RCU Splat

Lockdep-RCU was added to the Linux kernel in early 2010 (http://lwn.net/Articles/371986/). This facility checks for some common misuses of the RCU API, most notably using one of the rcu_dereference() family to access an RCU-protected pointer without the proper protection. When such misuse is detected, an lockdep-RCU splat is emitted.

The usual cause of a lockdep-RCU splat is someone accessing an RCU-protected data structure without either (1) being in the right kind of RCU read-side critical section or (2) holding the right update-side lock. This problem can therefore be serious: it might result in random memory overwriting or worse. There can of course be false positives, this being the real world and all that.

So let's look at an example RCU lockdep splat from 3.0-rc5, one that has long since been fixed:

```
=============================
WARNING: suspicious RCU usage
```

```
-----------------------------
block/cfq-iosched.c:2776 suspicious rcu_dereference_protected() usage!
```

other info that might help us debug this:

```
rcu_scheduler_active = 1, debug_locks = 0
3 locks held by scsi_scan_6/1552:
#0:  (&shost->scan_mutex){+.+.}, at: [<ffffffff8145efca>]
scsi_scan_host_selected+0x5a/0x150
#1:  (&eq->sysfs_lock){+.+.}, at: [<ffffffff812a5032>]
elevator_exit+0x22/0x60
#2:  (&(&q->__queue_lock)->rlock){-.-.}, at: [<ffffffff812b6233>]
cfq_exit_queue+0x43/0x190

stack backtrace:
Pid: 1552, comm: scsi_scan_6 Not tainted 3.0.0-rc5 #17
Call Trace:
[<ffffffff810abb9b>] lockdep_rcu_dereference+0xbb/0xc0
[<ffffffff812b6139>] __cfq_exit_single_io_context+0xe9/0x120
[<ffffffff812b626c>] cfq_exit_queue+0x7c/0x190
[<ffffffff812a5046>] elevator_exit+0x36/0x60
[<ffffffff812a802a>] blk_cleanup_queue+0x4a/0x60
[<ffffffff8145cc09>] scsi_free_queue+0x9/0x10
[<ffffffff81460944>] __scsi_remove_device+0x84/0xd0
[<ffffffff8145dca3>] scsi_probe_and_add_lun+0x353/0xb10
[<ffffffff817da069>] ? error_exit+0x29/0xb0
[<ffffffff817d98ed>] ? _raw_spin_unlock_irqrestore+0x3d/0x80
[<ffffffff8145e722>] __scsi_scan_target+0x112/0x680
[<ffffffff812c690d>] ? trace_hardirqs_off_thunk+0x3a/0x3c
[<ffffffff817da069>] ? error_exit+0x29/0xb0
[<ffffffff812bcc60>] ? kobject_del+0x40/0x40
[<ffffffff8145ed16>] scsi_scan_channel+0x86/0xb0
[<ffffffff8145f0b0>] scsi_scan_host_selected+0x140/0x150
[<ffffffff8145f149>] do_scsi_scan_host+0x89/0x90
[<ffffffff8145f170>] do_scan_async+0x20/0x160
[<ffffffff8145f150>] ? do_scsi_scan_host+0x90/0x90
[<ffffffff810975b6>] kthread+0xa6/0xb0
[<ffffffff817db154>] kernel_thread_helper+0x4/0x10
[<ffffffff81066430>] ? finish_task_switch+0x80/0x110
[<ffffffff817d9c04>] ? retint_restore_args+0xe/0xe
[<ffffffff81097510>] ? __kthread_init_worker+0x70/0x70
[<ffffffff817db150>] ? gs_change+0xb/0xb
```

Line 2776 of block/cfq-iosched.c in v3.0-rc5 is as follows:

```
if (rcu_dereference(ioc->ioc_data) == cic) {
```

This form says that it must be in a plain vanilla RCU read-side critical section, but the "other info" list above shows that this is not the case. Instead, we hold three locks, one of which might be RCU related. And maybe that lock really does protect this reference. If so, the fix is to inform RCU, perhaps by changing __cfq_exit_single_io_context() to take the struct request_queue "q" from cfq_exit_queue() as an argument, which would permit us to invoke rcu_dereference_protected as follows:

```
if (rcu_dereference_protected(ioc->ioc_data,
                              lockdep_is_held(&q->queue_lock)) == cic) {
```

With this change, there would be no lockdep-RCU splat emitted if this code was invoked either from within an RCU read-side critical section or with the ->queue_lock held. In particular, this would have suppressed the above lockdep-RCU splat because ->queue_lock is held (see #2 in the list above).

On the other hand, perhaps we really do need an RCU read-side critical section. In this case, the critical section must span the use of the return value from rcu_dereference(), or at least until there is some reference count incremented or some such. One way to handle this is to add rcu_read_lock() and rcu_read_unlock() as follows:

```
rcu_read_lock();
if (rcu_dereference(ioc->ioc_data) == cic) {
        spin_lock(&ioc->lock);
        rcu_assign_pointer(ioc->ioc_data, NULL);
        spin_unlock(&ioc->lock);
}
rcu_read_unlock();
```

With this change, the rcu_dereference() is always within an RCU read-side critical section, which again would have suppressed the above lockdep-RCU splat.

But in this particular case, we don't actually dereference the pointer returned from rcu_dereference(). Instead, that pointer is just compared to the cic pointer, which means that the rcu_dereference() can be replaced by `rcu_access_pointer()` as follows:

```
if (rcu_access_pointer(ioc->ioc_data) == cic) {
```

Because it is legal to invoke `rcu_access_pointer()` without protection, this change would also suppress the above lockdep-RCU splat.

4.5.4 RCU and Unloadable Modules

[Originally published in LWN Jan. 14, 2007: http://lwn.net/Articles/217484/]

RCU updaters sometimes use call_rcu() to initiate an asynchronous wait for a grace period to elapse. This primitive takes a pointer to an rcu_head struct placed within the RCU-protected data structure and another pointer to a function that may be invoked later to free that structure. Code to delete an element p from the linked list from IRQ context might then be as follows:

```
list_del_rcu(p);
call_rcu(&p->rcu, p_callback);
```

Since call_rcu() never blocks, this code can safely be used from within IRQ context. The function p_callback() might be defined as follows:

```
static void p_callback(struct rcu_head *rp)
{
```

```
        struct pstruct *p = container_of(rp, struct pstruct, rcu);

        kfree(p);
}
```

Unloading Modules That Use call_rcu()

But what if the p_callback() function is defined in an unloadable module?

If we unload the module while some RCU callbacks are pending, the CPUs executing these callbacks are going to be severely disappointed when they are later invoked, as fancifully depicted at http://lwn.net/images/ns/kernel/rcu-drop.jpg.

We could try placing a synchronize_rcu() in the module-exit code path, but this is not sufficient. Although synchronize_rcu() does wait for a grace period to elapse, it does not wait for the callbacks to complete.

One might be tempted to try several back-to-back synchronize_rcu() calls, but this is still not guaranteed to work. If there is a very heavy RCU-callback load, then some of the callbacks might be deferred in order to allow other processing to proceed. For but one example, such deferral is required in realtime kernels in order to avoid excessive scheduling latencies.

rcu_barrier()

This situation can be handled by the *rcu_barrier()* primitive. Rather than waiting for a grace period to elapse, *rcu_barrier()* waits for all outstanding RCU callbacks to complete. Please note that *rcu_barrier()* does **not** imply synchronize_rcu(), in particular, if there are no RCU callbacks queued anywhere, *rcu_barrier()* is within its rights to return immediately, without waiting for anything, let alone a grace period.

Pseudo-code using *rcu_barrier()* is as follows:

1. Prevent any new RCU callbacks from being posted.
2. Execute *rcu_barrier()*.
3. Allow the module to be unloaded.

There is also an *srcu_barrier()* function for SRCU, and you of course must match the flavor of *srcu_barrier()* with that of *call_srcu()*. If your module uses multiple srcu_struct structures, then it must also use multiple invocations of *srcu_barrier()* when unloading that module. For example, if it uses call_rcu(), *call_srcu()* on srcu_struct_1, and *call_srcu()* on srcu_struct_2, then the following three lines of code will be required when unloading:

```
rcu_barrier();
srcu_barrier(&srcu_struct_1);
srcu_barrier(&srcu_struct_2);
```

If latency is of the essence, workqueues could be used to run these three functions concurrently.

An ancient version of the rcutorture module makes use of *rcu_barrier()* in its exit function as follows:

```
static void
rcu_torture_cleanup(void)
{
  int i;

  fullstop = 1;
  if (shuffler_task != NULL) {
    VERBOSE_PRINTK_STRING("Stopping rcu_torture_shuffle task");
    kthread_stop(shuffler_task);
  }
  shuffler_task = NULL;

  if (writer_task != NULL) {
    VERBOSE_PRINTK_STRING("Stopping rcu_torture_writer task");
    kthread_stop(writer_task);
  }
  writer_task = NULL;

  if (reader_tasks != NULL) {
    for (i = 0; i < nrealreaders; i++) {
      if (reader_tasks[i] != NULL) {
        VERBOSE_PRINTK_STRING(
          "Stopping rcu_torture_reader task");
        kthread_stop(reader_tasks[i]);
      }
      reader_tasks[i] = NULL;
    }
    kfree(reader_tasks);
    reader_tasks = NULL;
  }
  rcu_torture_current = NULL;

  if (fakewriter_tasks != NULL) {
    for (i = 0; i < nfakewriters; i++) {
      if (fakewriter_tasks[i] != NULL) {
        VERBOSE_PRINTK_STRING(
          "Stopping rcu_torture_fakewriter task");
        kthread_stop(fakewriter_tasks[i]);
      }
      fakewriter_tasks[i] = NULL;
    }
    kfree(fakewriter_tasks);
    fakewriter_tasks = NULL;
  }

  if (stats_task != NULL) {
    VERBOSE_PRINTK_STRING("Stopping rcu_torture_stats task");
    kthread_stop(stats_task);
  }
  stats_task = NULL;
```

```
51
52     /* Wait for all RCU callbacks to fire. */
53     rcu_barrier();
54
55     rcu_torture_stats_print(); /* -After- the stats thread is stopped! */
56
57     if (cur_ops->cleanup != NULL)
58       cur_ops->cleanup();
59     if (atomic_read(&n_rcu_torture_error))
60       rcu_torture_print_module_parms("End of test: FAILURE");
61     else
62       rcu_torture_print_module_parms("End of test: SUCCESS");
63   }
```

Line 6 sets a global variable that prevents any RCU callbacks from re-posting themselves. This will not be necessary in most cases, since RCU callbacks rarely include calls to call_rcu(). However, the rcutorture module is an exception to this rule, and therefore needs to set this global variable.

Lines 7-50 stop all the kernel tasks associated with the rcutorture module. Therefore, once execution reaches line 53, no more rcutorture RCU callbacks will be posted. The *rcu_barrier()* call on line 53 waits for any pre-existing callbacks to complete.

Then lines 55-62 print status and do operation-specific cleanup, and then return, permitting the module-unload operation to be completed.

Quick Quiz #1:
Is there any other situation where *rcu_barrier()* might be required?

Answer to Quick Quiz #1

Your module might have additional complications. For example, if your module invokes call_rcu() from timers, you will need to first refrain from posting new timers, cancel (or wait for) all the already-posted timers, and only then invoke *rcu_barrier()* to wait for any remaining RCU callbacks to complete.

Of course, if your module uses call_rcu(), you will need to invoke *rcu_barrier()* before unloading. Similarly, if your module uses *call_srcu()*, you will need to invoke *srcu_barrier()* before unloading, and on the same srcu_struct structure. If your module uses call_rcu() **and** *call_srcu()*, then (as noted above) you will need to invoke *rcu_barrier()* **and** *srcu_barrier()*.

Implementing rcu_barrier()

Dipankar Sarma's implementation of *rcu_barrier()* makes use of the fact that RCU callbacks are never reordered once queued on one of the per-CPU queues. His implementation queues an RCU callback on each of the per-CPU callback queues, and then waits until they have all started executing, at which point, all earlier RCU callbacks are guaranteed to have completed.

The original code for *rcu_barrier()* was roughly as follows:

```
void rcu_barrier(void)
{
  BUG_ON(in_interrupt());
```

```
4    /* Take cpucontrol mutex to protect against CPU hotplug */
5    mutex_lock(&rcu_barrier_mutex);
6    init_completion(&rcu_barrier_completion);
7    atomic_set(&rcu_barrier_cpu_count, 1);
8    on_each_cpu(rcu_barrier_func, NULL, 0, 1);
9    if (atomic_dec_and_test(&rcu_barrier_cpu_count))
10     complete(&rcu_barrier_completion);
11   wait_for_completion(&rcu_barrier_completion);
12   mutex_unlock(&rcu_barrier_mutex);
13 }
```

Line 3 verifies that the caller is in process context, and lines 5 and 12 use rcu_barrier_mutex to ensure that only one *rcu_barrier()* is using the global completion and counters at a time, which are initialized on lines 6 and 7. Line 8 causes each CPU to invoke rcu_barrier_func(), which is shown below. Note that the final "1" in on_each_cpu()'s argument list ensures that all the calls to rcu_barrier_func() will have completed before on_each_cpu() returns. Line 9 removes the initial count from rcu_barrier_cpu_count, and if this count is now zero, line 10 finalizes the completion, which prevents line 11 from blocking. Either way, line 11 then waits (if needed) for the completion.

Quick Quiz #2:

Why doesn't line 8 initialize rcu_barrier_cpu_count to zero, thereby avoiding the need for lines 9 and 10?

Answer to Quick Quiz #2

This code was rewritten in 2008 and several times thereafter, but this still gives the general idea.

The rcu_barrier_func() runs on each CPU, where it invokes call_rcu() to post an RCU callback, as follows:

```
1  static void rcu_barrier_func(void *notused)
2  {
3    int cpu = smp_processor_id();
4    struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
5    struct rcu_head *head;
6
7    head = &rdp->barrier;
8    atomic_inc(&rcu_barrier_cpu_count);
9    call_rcu(head, rcu_barrier_callback);
10 }
```

Lines 3 and 4 locate RCU's internal per-CPU rcu_data structure, which contains the struct rcu_head that needed for the later call to call_rcu(). Line 7 picks up a pointer to this struct rcu_head, and line 8 increments the global counter. This counter will later be decremented by the callback. Line 9 then registers the rcu_barrier_callback() on the current CPU's queue.

The rcu_barrier_callback() function simply atomically decrements the rcu_barrier_cpu_count variable and finalizes the completion when it reaches zero, as follows:

```
1 static void rcu_barrier_callback(struct rcu_head *notused)
2 {
3   if (atomic_dec_and_test(&rcu_barrier_cpu_count))
```

```
    complete(&rcu_barrier_completion);
}
```

Quick Quiz #3:
What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in *rcu_barrier()* returning prematurely?

Answer to Quick Quiz #3

The current *rcu_barrier()* implementation is more complex, due to the need to avoid disturbing idle CPUs (especially on battery-powered systems) and the need to minimally disturb non-idle CPUs in real-time systems. In addition, a great many optimizations have been applied. However, the code above illustrates the concepts.

rcu_barrier() Summary

The *rcu_barrier()* primitive is used relatively infrequently, since most code using RCU is in the core kernel rather than in modules. However, if you are using RCU from an unloadable module, you need to use *rcu_barrier()* so that your module may be safely unloaded.

Answers to Quick Quizzes

Quick Quiz #1:
Is there any other situation where *rcu_barrier()* might be required?

Answer:
Interestingly enough, *rcu_barrier()* was not originally implemented for module unloading. Nikita Danilov was using RCU in a filesystem, which resulted in a similar situation at filesystem-unmount time. Dipankar Sarma coded up *rcu_barrier()* in response, so that Nikita could invoke it during the filesystem-unmount process.

Much later, yours truly hit the RCU module-unload problem when implementing rcutorture, and found that *rcu_barrier()* solves this problem as well.

Back to Quick Quiz #1

Quick Quiz #2:
Why doesn't line 8 initialize rcu_barrier_cpu_count to zero, thereby avoiding the need for lines 9 and 10?

Answer:
Suppose that the on_each_cpu() function shown on line 8 was delayed, so that CPU 0's rcu_barrier_func() executed and the corresponding grace period elapsed, all before CPU 1's rcu_barrier_func() started executing. This would result in rcu_barrier_cpu_count being decremented to zero, so that line 11's wait_for_completion() would return immediately, failing to wait for CPU 1's callbacks to be invoked.

Note that this was not a problem when the *rcu_barrier()* code was first added back in 2005. This is because on_each_cpu() disables preemption, which acted as an RCU read-side critical section, thus preventing CPU 0's grace period from completing until on_each_cpu() had dealt with all of the CPUs. However, with the advent of preemptible

RCU, *rcu_barrier()* no longer waited on nonpreemptible regions of code in preemptible kernels, that being the job of the new rcu_barrier_sched() function.

However, with the RCU flavor consolidation around v4.20, this possibility was once again ruled out, because the consolidated RCU once again waits on nonpreemptible regions of code.

Nevertheless, that extra count might still be a good idea. Relying on these sort of accidents of implementation can result in later surprise bugs when the implementation changes.

Back to Quick Quiz #2

Quick Quiz #3:

What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in *rcu_barrier()* returning prematurely?

Answer:

This cannot happen. The reason is that on_each_cpu() has its last argument, the wait flag, set to "1". This flag is passed through to smp_call_function() and further to smp_call_function_on_cpu(), causing this latter to spin until the cross-CPU invocation of rcu_barrier_func() has completed. This by itself would prevent a grace period from completing on non-CONFIG_PREEMPTION kernels, since each CPU must undergo a context switch (or other quiescent state) before the grace period can complete. However, this is of no use in CONFIG_PREEMPTION kernels.

Therefore, on_each_cpu() disables preemption across its call to smp_call_function() and also across the local call to rcu_barrier_func(). Because recent RCU implementations treat preemption-disabled regions of code as RCU read-side critical sections, this prevents grace periods from completing. This means that all CPUs have executed rcu_barrier_func() before the first rcu_barrier_callback() can possibly execute, in turn preventing rcu_barrier_cpu_count from prematurely reaching zero.

But if on_each_cpu() ever decides to forgo disabling preemption, as might well happen due to real-time latency considerations, initializing rcu_barrier_cpu_count to one will save the day.

Back to Quick Quiz #3

4.5.5 PROPER CARE AND FEEDING OF RETURN VALUES FROM rcu_dereference()

Most of the time, you can use values from rcu_dereference() or one of the similar primitives without worries. Dereferencing (prefix "*"), field selection ("->"), assignment ("="), address-of ("&"), addition and subtraction of constants, and casts all work quite naturally and safely.

It is nevertheless possible to get into trouble with other operations. Follow these rules to keep your RCU code working properly:

- You must use one of the rcu_dereference() family of primitives to load an RCU-protected pointer, otherwise CONFIG_PROVE_RCU will complain. Worse yet, your code can see random memory-corruption bugs due to games that compilers and DEC Alpha can play. Without one of the rcu_dereference() primitives, compilers can reload the value, and won't your code have fun with two different values for a single pointer! Without rcu_dereference(), DEC Alpha can load a pointer, dereference that pointer, and return data preceding initialization that preceded the store of the pointer. (As noted later, in recent kernels READ_ONCE() also prevents DEC Alpha from playing these tricks.)

In addition, the volatile cast in rcu_dereference() prevents the compiler from deducing the resulting pointer value. Please see the section entitled "EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH" for an example where the compiler can in fact deduce the exact value of the pointer, and thus cause misordering.

- In the special case where data is added but is never removed while readers are accessing the structure, READ_ONCE() may be used instead of rcu_dereference(). In this case, use of READ_ONCE() takes on the role of the lockless_dereference() primitive that was removed in v4.15.
- You are only permitted to use rcu_dereference() on pointer values. The compiler simply knows too much about integral values to trust it to carry dependencies through integer operations. There are a very few exceptions, namely that you can temporarily cast the pointer to uintptr_t in order to:
  - Set bits and clear bits down in the must-be-zero low-order bits of that pointer. This clearly means that the pointer must have alignment constraints, for example, this does *not* work in general for char* pointers.
  - XOR bits to translate pointers, as is done in some classic buddy-allocator algorithms.

  It is important to cast the value back to pointer before doing much of anything else with it.
- Avoid cancellation when using the "+" and "-" infix arithmetic operators. For example, for a given variable "x", avoid "(x-(uintptr_t)x)" for char* pointers. The compiler is within its rights to substitute zero for this sort of expression, so that subsequent accesses no longer depend on the rcu_dereference(), again possibly resulting in bugs due to misordering.

  Of course, if "p" is a pointer from rcu_dereference(), and "a" and "b" are integers that happen to be equal, the expression "p+a-b" is safe because its value still necessarily depends on the rcu_dereference(), thus maintaining proper ordering.
- If you are using RCU to protect JITed functions, so that the "()" function-invocation operator is applied to a value obtained (directly or indirectly) from rcu_dereference(), you may need to interact directly with the hardware to flush instruction caches. This issue arises on some systems when a newly JITed function is using the same memory that was used by an earlier JITed function.
- Do not use the results from relational operators ("==", "!=", ">", ">=", "<", or "<=") when dereferencing. For example, the following (quite strange) code is buggy:

```
int *p;
int *q;

...

p = rcu_dereference(gp)
q = &global_q;
q += p > &oom_p;
r1 = *q;  /* BUGGY!!! */
```

As before, the reason this is buggy is that relational operators are often compiled using branches. And as before, although weak-memory machines such as ARM or PowerPC do order stores after such branches, but can speculate loads, which can again result in misordering bugs.

- Be very careful about comparing pointers obtained from rcu_dereference() against non-NULL values. As Linus Torvalds explained, if the two pointers are equal, the compiler could substitute the pointer you are comparing against for the pointer obtained from rcu_dereference(). For example:

```
p = rcu_dereference(gp);
if (p == &default_struct)
        do_default(p->a);
```

  Because the compiler now knows that the value of "p" is exactly the address of the variable "default_struct", it is free to transform this code into the following:

```
p = rcu_dereference(gp);
if (p == &default_struct)
        do_default(default_struct.a);
```

  On ARM and Power hardware, the load from "default_struct.a" can now be speculated, such that it might happen before the rcu_dereference(). This could result in bugs due to misordering.

  However, comparisons are OK in the following cases:

  - The comparison was against the NULL pointer. If the compiler knows that the pointer is NULL, you had better not be dereferencing it anyway. If the comparison is non-equal, the compiler is none the wiser. Therefore, it is safe to compare pointers from rcu_dereference() against NULL pointers.
  - The pointer is never dereferenced after being compared. Since there are no subsequent dereferences, the compiler cannot use anything it learned from the comparison to reorder the non-existent subsequent dereferences. This sort of comparison occurs frequently when scanning RCU-protected circular linked lists.

    Note that if the pointer comparison is done outside of an RCU read-side critical section, and the pointer is never dereferenced, `rcu_access_pointer()` should be used in place of rcu_dereference(). In most cases, it is best to avoid accidental dereferences by testing the `rcu_access_pointer()` return value directly, without assigning it to a variable.

    Within an RCU read-side critical section, there is little reason to use `rcu_access_pointer()`.
  - The comparison is against a pointer that references memory that was initialized "a long time ago." The reason this is safe is that even if misordering occurs, the misordering will not affect the accesses that follow the comparison. So exactly how long ago is "a long time ago"? Here are some possibilities:
    * Compile time.
    * Boot time.
    * Module-init time for module code.

    * Prior to kthread creation for kthread code.
    * During some prior acquisition of the lock that we now hold.
    * Before mod_timer() time for a timer handler.

  There are many other possibilities involving the Linux kernel's wide array of primitives that cause code to be invoked at a later time.

- The pointer being compared against also came from rcu_dereference(). In this case, both pointers depend on one rcu_dereference() or another, so you get proper ordering either way.

  That said, this situation can make certain RCU usage bugs more likely to happen. Which can be a good thing, at least if they happen during testing. An example of such an RCU usage bug is shown in the section titled "EXAMPLE OF AMPLIFIED RCU-USAGE BUG".

- All of the accesses following the comparison are stores, so that a control dependency preserves the needed ordering. That said, it is easy to get control dependencies wrong. Please see the "CONTROL DEPENDENCIES" section of Documentation/memory-barriers.txt for more details.

- The pointers are not equal *and* the compiler does not have enough information to deduce the value of the pointer. Note that the volatile cast in rcu_dereference() will normally prevent the compiler from knowing too much.

  However, please note that if the compiler knows that the pointer takes on only one of two values, a not-equal comparison will provide exactly the information that the compiler needs to deduce the value of the pointer.

• Disable any value-speculation optimizations that your compiler might provide, especially if you are making use of feedback-based optimizations that take data collected from prior runs. Such value-speculation optimizations reorder operations by design.

  There is one exception to this rule: Value-speculation optimizations that leverage the branch-prediction hardware are safe on strongly ordered systems (such as x86), but not on weakly ordered systems (such as ARM or Power). Choose your compiler command-line options wisely!

EXAMPLE OF AMPLIFIED RCU-USAGE BUG

Because updaters can run concurrently with RCU readers, RCU readers can see stale and/or inconsistent values. If RCU readers need fresh or consistent values, which they sometimes do, they need to take proper precautions. To see this, consider the following code fragment:

```
struct foo {
        int a;
        int b;
        int c;
};
struct foo *gp1;
struct foo *gp2;

void updater(void)
{
```

```
        struct foo *p;

        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        rcu_assign_pointer(gp1, p);
        p->b = 143;
        p->c = 144;
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
        struct foo *q;
        int r1, r2;

        rcu_read_lock();
        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        r1 = p->b;  /* Guaranteed to get 143. */
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
                r2 = p->c; /* Could get 44 on weakly order system. */
        } else {
                r2 = p->c - r1; /* Unconditional access to p->c. */
        }
        rcu_read_unlock();
        do_something_with(r1, r2);
}
```

You might be surprised that the outcome (r1 == 143 && r2 == 44) is possible, but you should not be. After all, the updater might have been invoked a second time between the time reader() loaded into “r1” and the time that it loaded into “r2”. The fact that this same result can occur due to some reordering from the compiler and CPUs is beside the point.

But suppose that the reader needs a consistent view?

Then one approach is to use locking, for example, as follows:

```
struct foo {
        int a;
        int b;
        int c;
        spinlock_t lock;
};
struct foo *gp1;
```

```
struct foo *gp2;

void updater(void)
{
        struct foo *p;

        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        spin_lock(&p->lock);
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp1, p);
        spin_lock(&p->lock);
        p->b = 143;
        p->c = 144;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
        struct foo *q;
        int r1, r2;

        rcu_read_lock();
        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        spin_lock(&p->lock);
        r1 = p->b;  /* Guaranteed to get 143. */
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
                r2 = p->c; /* Locking guarantees r2 == 144. */
        } else {
                spin_lock(&q->lock);
                r2 = q->c - r1;
                spin_unlock(&q->lock);
        }
        rcu_read_unlock();
        spin_unlock(&p->lock);
        do_something_with(r1, r2);
}
```

As always, use the right tool for the job!

EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH

If a pointer obtained from rcu_dereference() compares not-equal to some other pointer, the compiler normally has no clue what the value of the first pointer might be. This lack of knowledge prevents the compiler from carrying out optimizations that otherwise might destroy the ordering guarantees that RCU depends on. And the volatile cast in rcu_dereference() should prevent the compiler from guessing the value.

But without rcu_dereference(), the compiler knows more than you might expect. Consider the following code fragment:

```
struct foo {
        int a;
        int b;
};
static struct foo variable1;
static struct foo variable2;
static struct foo *gp = &variable1;

void updater(void)
{
        initialize_foo(&variable2);
        rcu_assign_pointer(gp, &variable2);
        /*
         * The above is the only store to gp in this translation unit,
         * and the address of gp is not exported in any way.
         */
}

int reader(void)
{
        struct foo *p;

        p = gp;
        barrier();
        if (p == &variable1)
                return p->a; /* Must be variable1.a. */
        else
                return p->b; /* Must be variable2.b. */
}
```

Because the compiler can see all stores to "gp", it knows that the only possible values of "gp" are "variable1" on the one hand and "variable2" on the other. The comparison in reader() therefore tells the compiler the exact value of "p" even in the not-equals case. This allows the compiler to make the return values independent of the load from "gp", in turn destroying the ordering between this load and the loads of the return values. This can result in "p->b" returning pre-initialization garbage values on weakly ordered systems.

In short, rcu_dereference() is *not* optional when you are going to dereference the resulting pointer.

WHICH MEMBER OF THE rcu_dereference() FAMILY SHOULD YOU USE?

First, please avoid using rcu_dereference_raw() and also please avoid using *rcu_dereference_check()* and *rcu_dereference_protected()* with a second argument with a constant value of 1 (or true, for that matter). With that caution out of the way, here is some guidance for which member of the rcu_dereference() to use in various situations:

1. If the access needs to be within an RCU read-side critical section, use rcu_dereference(). With the new consolidated RCU flavors, an RCU read-side critical section is entered using rcu_read_lock(), anything that disables bottom halves, anything that disables interrupts, or anything that disables preemption.

2. If the access might be within an RCU read-side critical section on the one hand, or protected by (say) my_lock on the other, use *rcu_dereference_check()*, for example:

   ```
   p1 = rcu_dereference_check(p->rcu_protected_pointer,
                              lockdep_is_held(&my_lock));
   ```

3. If the access might be within an RCU read-side critical section on the one hand, or protected by either my_lock or your_lock on the other, again use *rcu_dereference_check()*, for example:

   ```
   p1 = rcu_dereference_check(p->rcu_protected_pointer,
                              lockdep_is_held(&my_lock) ||
                              lockdep_is_held(&your_lock));
   ```

4. If the access is on the update side, so that it is always protected by my_lock, use *rcu_dereference_protected()*:

   ```
   p1 = rcu_dereference_protected(p->rcu_protected_pointer,
                                  lockdep_is_held(&my_lock));
   ```

   This can be extended to handle multiple locks as in #3 above, and both can be extended to check other conditions as well.

5. If the protection is supplied by the caller, and is thus unknown to this code, that is the rare case when rcu_dereference_raw() is appropriate. In addition, rcu_dereference_raw() might be appropriate when the lockdep expression would be excessively complex, except that a better approach in that case might be to take a long hard look at your synchronization design. Still, there are data-locking cases where any one of a very large number of locks or reference counters suffices to protect the pointer, so rcu_dereference_raw() does have its place.

   However, its place is probably quite a bit smaller than one might expect given the number of uses in the current kernel. Ditto for its synonym, rcu_dereference_check(… , 1), and its close relative, rcu_dereference_protected(… , 1).

SPARSE CHECKING OF RCU-PROTECTED POINTERS

The sparse static-analysis tool checks for non-RCU access to RCU-protected pointers, which can result in "interesting" bugs due to compiler optimizations involving invented loads and perhaps also load tearing. For example, suppose someone mistakenly does something like this:

```
p = q->rcu_protected_pointer;
do_something_with(p->a);
do_something_else_with(p->b);
```

If register pressure is high, the compiler might optimize "p" out of existence, transforming the code to something like this:

```
do_something_with(q->rcu_protected_pointer->a);
do_something_else_with(q->rcu_protected_pointer->b);
```

This could fatally disappoint your code if q->rcu_protected_pointer changed in the meantime. Nor is this a theoretical problem: Exactly this sort of bug cost Paul E. McKenney (and several of his innocent colleagues) a three-day weekend back in the early 1990s.

Load tearing could of course result in dereferencing a mashup of a pair of pointers, which also might fatally disappoint your code.

These problems could have been avoided simply by making the code instead read as follows:

```
p = rcu_dereference(q->rcu_protected_pointer);
do_something_with(p->a);
do_something_else_with(p->b);
```

Unfortunately, these sorts of bugs can be extremely hard to spot during review. This is where the sparse tool comes into play, along with the "__rcu" marker. If you mark a pointer declaration, whether in a structure or as a formal parameter, with "__rcu", which tells sparse to complain if this pointer is accessed directly. It will also cause sparse to complain if a pointer not marked with "__rcu" is accessed using rcu_dereference() and friends. For example, ->rcu_protected_pointer might be declared as follows:

```
struct foo __rcu *rcu_protected_pointer;
```

Use of "__rcu" is opt-in. If you choose not to use it, then you should ignore the sparse warnings.

4.5.6 What is RCU? -- "Read, Copy, Update"

Please note that the "What is RCU?" LWN series is an excellent place to start learning about RCU:

1. What is RCU, Fundamentally? https://lwn.net/Articles/262464/
2. What is RCU? Part 2: Usage https://lwn.net/Articles/263130/
3. RCU part 3: the RCU API https://lwn.net/Articles/264090/
4. The RCU API, 2010 Edition https://lwn.net/Articles/418853/
   2010 Big API Table https://lwn.net/Articles/419086/
5. The RCU API, 2014 Edition https://lwn.net/Articles/609904/

2014 Big API Table https://lwn.net/Articles/609973/

6. The RCU API, 2019 Edition https://lwn.net/Articles/777036/

   2019 Big API Table https://lwn.net/Articles/777165/

For those preferring video:

1. Unraveling RCU Mysteries: Fundamentals https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries
2. Unraveling RCU Mysteries: Additional Use Cases https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries-additional-use-cases

What is RCU?

RCU is a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort that is optimized for read-mostly situations. Although RCU is actually quite simple, making effective use of it requires you to think differently about your code. Another part of the problem is the mistaken assumption that there is "one true way" to describe and to use RCU. Instead, the experience has been that different people must take different paths to arrive at an understanding of RCU, depending on their experiences and use cases. This document provides several different paths, as follows:

People who prefer starting with a conceptual overview should focus on Section 1, though most readers will profit by reading this section at some point. People who prefer to start with an API that they can then experiment with should focus on Section 2. People who prefer to start with example uses should focus on Sections 3 and 4. People who need to understand the RCU implementation should focus on Section 5, then dive into the kernel source code. People who reason best by analogy should focus on Section 6. Section 7 serves as an index to the docbook API documentation, and Section 8 is the traditional answer key.

So, start with the section that makes the most sense to you and your preferred method of learning. If you need to know everything about everything, feel free to read the whole thing -- but if you are really that type of person, you have perused the source code and will therefore never need this document anyway. ;-)

1. RCU OVERVIEW

The basic idea behind RCU is to split updates into "removal" and "reclamation" phases. The removal phase removes references to data items within a data structure (possibly by replacing them with references to new versions of these data items), and can run concurrently with readers. The reason that it is safe to run the removal phase concurrently with readers is the semantics of modern CPUs guarantee that readers will see either the old or the new version of the data structure rather than a partially updated reference. The reclamation phase does the work of reclaiming (e.g., freeing) the data items removed from the data structure during the removal phase. Because reclaiming data items can disrupt any readers concurrently referencing those data items, the reclamation phase must not start until readers no longer hold references to those data items.

Splitting the update into removal and reclamation phases permits the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the removal phase have completed, either by blocking until they finish or by registering a callback that is invoked after they finish. Only readers that are active during the removal phase need be considered, because any reader starting after the removal phase will be unable to gain a reference to the removed data items, and therefore cannot be disrupted by the reclamation phase.

So the typical RCU update sequence goes something like the following:

a. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.

b. Wait for all previous readers to complete their RCU read-side critical sections.

c. At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed (e.g., kfree()d).

Step (b) above is the key idea underlying RCU's deferred destruction. The ability to wait until all readers are done allows RCU readers to use much lighter-weight synchronization, in some cases, absolutely no synchronization at all. In contrast, in more conventional lock-based schemes, readers must use heavy-weight synchronization in order to prevent an updater from deleting the data structure out from under them. This is because lock-based updaters typically update data items in place, and must therefore exclude readers. In contrast, RCU-based updaters typically take advantage of the fact that writes to single aligned pointers are atomic on modern CPUs, allowing atomic insertion, removal, and replacement of data items in a linked structure without disrupting readers. Concurrent RCU readers can then continue accessing the old versions, and can dispense with the atomic operations, memory barriers, and communications cache misses that are so expensive on present-day SMP computer systems, even in absence of lock contention.

In the three-step procedure shown above, the updater is performing both the removal and the reclamation step, but it is often helpful for an entirely different thread to do the reclamation, as is in fact the case in the Linux kernel's directory-entry cache (dcache). Even if the same thread performs both the update step (step (a) above) and the reclamation step (step (c) above), it is often helpful to think of them separately. For example, RCU readers and updaters need not communicate at all, but RCU provides implicit low-overhead communication between readers and reclaimers, namely, in step (b) above.

So how the heck can a reclaimer tell when a reader is done, given that readers are not doing any sort of synchronization operations??? Read on to learn about how RCU's API makes this easy.

2. WHAT IS RCU'S CORE API?

The core RCU API is quite small:

a. rcu_read_lock()

b. rcu_read_unlock()

c. synchronize_rcu() / call_rcu()

d. rcu_assign_pointer()

e. rcu_dereference()

There are many other members of the RCU API, but the rest can be expressed in terms of these five, though most implementations instead express synchronize_rcu() in terms of the call_rcu() callback API.

The five core RCU APIs are described below, the other 18 will be enumerated later. See the kernel docbook documentation for more info, or look directly at the function header comments.

rcu_read_lock()

void rcu_read_lock(void);

This temporal primitive is used by a reader to inform the reclaimer that the reader is entering an RCU read-side critical section. It is illegal to block while in an RCU read-side critical section, though kernels built with CONFIG_PREEMPT_RCU can preempt RCU read-side critical sections. Any RCU-protected data structure accessed during an RCU read-side critical section is guaranteed to remain unreclaimed for the full duration of that critical section. Reference counts may be used in conjunction with RCU to maintain longer-term references to data structures.

rcu_read_unlock()

void rcu_read_unlock(void);

This temporal primitives is used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.

synchronize_rcu()

void synchronize_rcu(void);

This temporal primitive marks the end of updater code and the beginning of reclaimer code. It does this by blocking until all pre-existing RCU read-side critical sections on all CPUs have completed. Note that synchronize_rcu() will **not** necessarily wait for any subsequent RCU read-side critical sections to complete. For example, consider the following sequence of events:

```
      CPU 0                  CPU 1                 CPU 2
----------------- ------------------------- ---------------
```

```
1.  rcu_read_lock()
2.                   enters synchronize_rcu()
3.                                                rcu_read_lock()
4.  rcu_read_unlock()
5.                    exits synchronize_rcu()
6.                                               rcu_read_unlock()
```

To reiterate, synchronize_rcu() waits only for ongoing RCU read-side critical sections to complete, not necessarily for any that begin after synchronize_rcu() is invoked.

Of course, synchronize_rcu() does not necessarily return **immediately** after the last pre-existing RCU read-side critical section completes. For one thing, there might well be scheduling delays. For another thing, many RCU implementations process requests in batches in order to improve efficiencies, which can further delay synchronize_rcu().

Since synchronize_rcu() is the API that must figure out when readers are done, its implementation is key to RCU. For RCU to be useful in all but the most read-intensive situations, synchronize_rcu()'s overhead must also be quite small.

The call_rcu() API is an asynchronous callback form of synchronize_rcu(), and is described in more detail in a later section. Instead of blocking, it registers a function and argument which are invoked after all ongoing RCU read-side critical sections have completed. This callback variant is particularly useful in situations where it is illegal to block or where update-side performance is critically important.

However, the call_rcu() API should not be used lightly, as use of the synchronize_rcu() API generally results in simpler code. In addition, the synchronize_rcu() API has the nice property of automatically limiting update rate should grace periods be delayed. This property results in system resilience in face of denial-of-service attacks. Code using call_rcu() should limit update rate in order to gain this same sort of resilience. See checklist.rst for some approaches to limiting the update rate.

rcu_assign_pointer()

void rcu_assign_pointer(p, typeof(p) v);

Yes, rcu_assign_pointer() **is** implemented as a macro, though it would be cool to be able to declare a function in this manner. (Compiler experts will no doubt disagree.)

The updater uses this spatial macro to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This is a spatial (as opposed to temporal) macro. It does not evaluate to an rvalue, but it does execute any memory-barrier instructions required for a given CPU architecture. Its ordering properties are that of a store-release operation.

Perhaps just as important, it serves to document (1) which pointers are protected by RCU and (2) the point at which a given structure becomes accessible to other CPUs. That said, rcu_assign_pointer() is most frequently used indirectly, via the _rcu list-manipulation primitives such as list_add_rcu().

rcu_dereference()

typeof(p) rcu_dereference(p);

Like rcu_assign_pointer(), rcu_dereference() must be implemented as a macro.

The reader uses the spatial rcu_dereference() macro to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. Note that rcu_dereference() does not actually dereference the pointer, instead, it protects the pointer for later dereferencing. It also executes any needed memory-barrier instructions for a given CPU architecture. Currently, only Alpha needs memory barriers within rcu_dereference() -- on other CPUs, it compiles to a volatile load.

Common coding practice uses rcu_dereference() to copy an RCU-protected pointer to a local variable, then dereferences this local variable, for example as follows:

```
p = rcu_dereference(head.next);
return p->data;
```

However, in this case, one could just as easily combine these into one statement:

```
return rcu_dereference(head.next)->data;
```

If you are going to be fetching multiple fields from the RCU-protected structure, using the local variable is of course preferred. Repeated rcu_dereference() calls look ugly, do not guarantee that the same pointer will be returned if an update happened while in the critical section, and incur unnecessary overhead on Alpha CPUs.

Note that the value returned by rcu_dereference() is valid only within the enclosing RCU read-side critical section[1]. For example, the following is **not** legal:

```
rcu_read_lock();
p = rcu_dereference(head.next);
rcu_read_unlock();
x = p->address; /* BUG!!! */
rcu_read_lock();
y = p->data;    /* BUG!!! */
rcu_read_unlock();
```

Holding a reference from one RCU read-side critical section to another is just as illegal as holding a reference from one lock-based critical section to another! Similarly, using a reference outside of the critical section in which it was acquired is just as illegal as doing so with normal locking.

As with rcu_assign_pointer(), an important function of rcu_dereference() is to document which pointers are protected by RCU, in particular, flagging a pointer that is subject to changing at any time, including immediately after the rcu_dereference().

[1] The variant *rcu_dereference_protected()* can be used outside of an RCU read-side critical section as long as the usage is protected by locks acquired by the update-side code. This variant avoids the lockdep warning that would happen when using (for example) rcu_dereference() without rcu_read_lock() protection. Using *rcu_dereference_protected()* also has the advantage of permitting compiler optimizations that rcu_dereference() must prohibit. The *rcu_dereference_protected()* variant takes a lockdep expression to indicate which locks must be acquired by the caller. If the indicated protection is not provided, a lockdep splat is emitted. See Design/Requirements/Requirements.rst and the API's code comments for more details and example usage.

> And, again like rcu_assign_pointer(), rcu_dereference() is typically used indirectly, via the _rcu list-manipulation primitives, such as list_for_each_entry_rcu()[2].

The following diagram shows how each API communicates among the reader, updater, and reclaimer.

```
rcu_assign_pointer()
                        +--------+
+---------------------->| reader |---------+
|                       +--------+         |
|                           |              |
|                           |              | Protect:
|                           |              | rcu_read_lock()
|                           |              | rcu_read_unlock()
|        rcu_dereference()  |              |
+---------+                 |              |
| updater |<----------------+              |
+---------+                                V
|                                    +-----------+
+----------------------------------->| reclaimer |
                                     +-----------+
  Defer:
  synchronize_rcu() & call_rcu()
```

The RCU infrastructure observes the temporal sequence of rcu_read_lock(), rcu_read_unlock(), synchronize_rcu(), and call_rcu() invocations in order to determine when (1) synchronize_rcu() invocations may return to their callers and (2) call_rcu() callbacks may be invoked. Efficient implementations of the RCU infrastructure make heavy use of batching in order to amortize their overhead over many uses of the corresponding APIs. The rcu_assign_pointer() and rcu_dereference() invocations communicate spatial changes via stores to and loads from the RCU-protected pointer in question.

There are at least three flavors of RCU usage in the Linux kernel. The diagram above shows the most common one. On the updater side, the rcu_assign_pointer(), synchronize_rcu() and call_rcu() primitives used are the same for all three flavors. However for protection (on the reader side), the primitives used vary depending on the flavor:

a. rcu_read_lock() / rcu_read_unlock() rcu_dereference()

b. *rcu_read_lock_bh()* / *rcu_read_unlock_bh()* local_bh_disable() / local_bh_enable() *rcu_dereference_bh()*

c. *rcu_read_lock_sched()* / *rcu_read_unlock_sched()* preempt_disable() / preempt_enable() local_irq_save() / local_irq_restore() hardirq enter / hardirq exit NMI enter / NMI exit *rcu_dereference_sched()*

These three flavors are used as follows:

a. RCU applied to normal data structures.

b. RCU applied to networking data structures that may be subjected to remote denial-of-service attacks.

[2] If the list_for_each_entry_rcu() instance might be used by update-side code as well as by RCU readers, then an additional lockdep expression can be added to its list of arguments. For example, given an additional "lock_is_held(&mylock)" argument, the RCU lockdep code would complain only if this instance was invoked outside of an RCU read-side critical section and without the protection of mylock.

c. RCU applied to scheduler and interrupt/NMI-handler tasks.

Again, most uses will be of (a). The (b) and (c) cases are important for specialized uses, but are relatively uncommon. The SRCU, RCU-Tasks, RCU-Tasks-Rude, and RCU-Tasks-Trace have similar relationships among their assorted primitives.

3. WHAT ARE SOME EXAMPLE USES OF CORE RCU API?

This section shows a simple use of the core RCU API to protect a global pointer to a dynamically allocated structure. More-typical uses of RCU may be found in listRCU.rst, arrayRCU.rst, and NMI-RCU.rst.

```
struct foo {
        int a;
        char b;
        long c;
};
DEFINE_SPINLOCK(foo_mutex);

struct foo __rcu *gbl_foo;

/*
 * Create a new struct foo that is the same as the one currently
 * pointed to by gbl_foo, except that field "a" is replaced
 * with "new_a".  Points gbl_foo to the new structure, and
 * frees up the old structure after a grace period.
 *
 * Uses rcu_assign_pointer() to ensure that concurrent readers
 * see the initialized version of the new structure.
 *
 * Uses synchronize_rcu() to ensure that any readers that might
 * have references to the old structure complete before freeing
 * the old structure.
 */
void foo_update_a(int new_a)
{
        struct foo *new_fp;
        struct foo *old_fp;

        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_
↪mutex));
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        synchronize_rcu();
        kfree(old_fp);
}
```

```
/*
 * Return the value of field "a" of the current gbl_foo
 * structure.  Use rcu_read_lock() and rcu_read_unlock()
 * to ensure that the structure does not get deleted out
 * from under us, and use rcu_dereference() to ensure that
 * we see the initialized version of the structure (important
 * for DEC Alpha and for people reading the code).
 */
int foo_get_a(void)
{
        int retval;

        rcu_read_lock();
        retval = rcu_dereference(gbl_foo)->a;
        rcu_read_unlock();
        return retval;
}
```

So, to sum up:

- Use rcu_read_lock() and rcu_read_unlock() to guard RCU read-side critical sections.
- Within an RCU read-side critical section, use rcu_dereference() to dereference RCU-protected pointers.
- Use some solid design (such as locks or semaphores) to keep concurrent updates from interfering with each other.
- Use rcu_assign_pointer() to update an RCU-protected pointer. This primitive protects concurrent readers from the updater, **not** concurrent updates from each other! You therefore still need to use locking (or something similar) to keep concurrent rcu_assign_pointer() primitives from interfering with each other.
- Use synchronize_rcu() **after** removing a data element from an RCU-protected data structure, but **before** reclaiming/freeing the data element, in order to wait for the completion of all RCU read-side critical sections that might be referencing that data item.

See checklist.rst for additional rules to follow when using RCU. And again, more-typical uses of RCU may be found in listRCU.rst, arrayRCU.rst, and NMI-RCU.rst.

4. WHAT IF MY UPDATING THREAD CANNOT BLOCK?

In the example above, foo_update_a() blocks until a grace period elapses. This is quite simple, but in some cases one cannot afford to wait so long -- there might be other high-priority work to be done.

In such cases, one uses call_rcu() rather than synchronize_rcu(). The call_rcu() API is as follows:

```
void call_rcu(struct rcu_head *head, rcu_callback_t func);
```

This function invokes func(head) after a grace period has elapsed. This invocation might happen from either softirq or process context, so the function is not permitted to block. The foo struct needs to have an rcu_head structure added, perhaps as follows:

```
struct foo {
        int a;
        char b;
        long c;
        struct rcu_head rcu;
};
```

The foo_update_a() function might then be written as follows:

```
/*
 * Create a new struct foo that is the same as the one currently
 * pointed to by gbl_foo, except that field "a" is replaced
 * with "new_a".  Points gbl_foo to the new structure, and
 * frees up the old structure after a grace period.
 *
 * Uses rcu_assign_pointer() to ensure that concurrent readers
 * see the initialized version of the new structure.
 *
 * Uses call_rcu() to ensure that any readers that might have
 * references to the old structure complete before freeing the
 * old structure.
 */
void foo_update_a(int new_a)
{
        struct foo *new_fp;
        struct foo *old_fp;

        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_
↪mutex));
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        call_rcu(&old_fp->rcu, foo_reclaim);
}
```

The foo_reclaim() function might appear as follows:

```
void foo_reclaim(struct rcu_head *rp)
{
        struct foo *fp = container_of(rp, struct foo, rcu);

        foo_cleanup(fp->a);

        kfree(fp);
}
```

The container_of() primitive is a macro that, given a pointer into a struct, the type of the struct, and the pointed-to field within the struct, returns a pointer to the beginning of the struct.

The use of call_rcu() permits the caller of foo_update_a() to immediately regain control, without needing to worry further about the old version of the newly updated element. It also clearly shows the RCU distinction between updater, namely foo_update_a(), and reclaimer, namely foo_reclaim().

The summary of advice is the same as for the previous section, except that we are now using call_rcu() rather than synchronize_rcu():

- Use call_rcu() **after** removing a data element from an RCU-protected data structure in order to register a callback function that will be invoked after the completion of all RCU read-side critical sections that might be referencing that data item.

If the callback for call_rcu() is not doing anything more than calling kfree() on the structure, you can use kfree_rcu() instead of call_rcu() to avoid having to write your own callback:

```
kfree_rcu(old_fp, rcu);
```

If the occasional sleep is permitted, the single-argument form may be used, omitting the rcu_head structure from struct foo.

> kfree_rcu_mightsleep(old_fp);

This variant almost never blocks, but might do so by invoking synchronize_rcu() in response to memory-allocation failure.

Again, see checklist.rst for additional rules governing the use of RCU.

5. WHAT ARE SOME SIMPLE IMPLEMENTATIONS OF RCU?

One of the nice things about RCU is that it has extremely simple "toy" implementations that are a good first step towards understanding the production-quality implementations in the Linux kernel. This section presents two such "toy" implementations of RCU, one that is implemented in terms of familiar locking primitives, and another that more closely resembles "classic" RCU. Both are way too simple for real-world use, lacking both functionality and performance. However, they are useful in getting a feel for how RCU works. See kernel/rcu/update.c for a production-quality implementation, and see:

> https://docs.google.com/document/d/1X0lThx8OK0ZgLMqVoXiR4ZrGURHrXK6NyLRbeXe3Xac/edit

for papers describing the Linux kernel RCU implementation. The OLS'01 and OLS'02 papers are a good introduction, and the dissertation provides more details on the current implementation as of early 2004.

5A. "TOY" IMPLEMENTATION #1: LOCKING

This section presents a "toy" RCU implementation that is based on familiar locking primitives. Its overhead makes it a non-starter for real-life use, as does its lack of scalability. It is also unsuitable for realtime use, since it allows scheduling latency to "bleed" from one read-side critical section to another. It also assumes recursive reader-writer locks: If you try this with non-recursive locks, and you allow nested rcu_read_lock() calls, you can deadlock.

However, it is probably the easiest implementation to relate to, so is a good starting point.

It is extremely simple:

```
static DEFINE_RWLOCK(rcu_gp_mutex);

void rcu_read_lock(void)
{
        read_lock(&rcu_gp_mutex);
}

void rcu_read_unlock(void)
{
        read_unlock(&rcu_gp_mutex);
}

void synchronize_rcu(void)
{
        write_lock(&rcu_gp_mutex);
        smp_mb__after_spinlock();
        write_unlock(&rcu_gp_mutex);
}
```

[You can ignore rcu_assign_pointer() and rcu_dereference() without missing much. But here are simplified versions anyway. And whatever you do, don't forget about them when submitting patches making use of RCU!]:

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_store_release(&(p), (v)); \
})

#define rcu_dereference(p) \
({ \
        typeof(p) _________p1 = READ_ONCE(p); \
        (_________p1); \
})
```

The rcu_read_lock() and rcu_read_unlock() primitive read-acquire and release a global reader-writer lock. The synchronize_rcu() primitive write-acquires this same lock, then releases it. This means that once synchronize_rcu() exits, all RCU read-side critical sections that were in progress before synchronize_rcu() was called are guaranteed to have completed -- there is no way that synchronize_rcu() would have been able to write-acquire the lock otherwise. The smp_mb__after_spinlock() promotes synchronize_rcu() to a full memory barrier in compliance with the "Memory-Barrier Guarantees" listed in:

> Design/Requirements/Requirements.rst

It is possible to nest rcu_read_lock(), since reader-writer locks may be recursively acquired. Note also that rcu_read_lock() is immune from deadlock (an important property of RCU). The reason for this is that the only thing that can block rcu_read_lock() is a synchronize_rcu(). But synchronize_rcu() does not acquire any locks while holding rcu_gp_mutex, so there can be no deadlock cycle.

Quick Quiz #1:

> Why is this argument naive? How could a deadlock occur when using this algorithm in a

real-world Linux kernel? How could this deadlock be avoided?

Answers to Quick Quiz

5B. "TOY" EXAMPLE #2: CLASSIC RCU

This section presents a "toy" RCU implementation that is based on "classic RCU". It is also short on performance (but only for updates) and on features such as hotplug CPU and the ability to run in CONFIG_PREEMPTION kernels. The definitions of rcu_dereference() and rcu_assign_pointer() are the same as those shown in the preceding section, so they are omitted.

```
void rcu_read_lock(void) { }

void rcu_read_unlock(void) { }

void synchronize_rcu(void)
{
        int cpu;

        for_each_possible_cpu(cpu)
                run_on(cpu);
}
```

Note that rcu_read_lock() and rcu_read_unlock() do absolutely nothing. This is the great strength of classic RCU in a non-preemptive kernel: read-side overhead is precisely zero, at least on non-Alpha CPUs. And there is absolutely no way that rcu_read_lock() can possibly participate in a deadlock cycle!

The implementation of synchronize_rcu() simply schedules itself on each CPU in turn. The run_on() primitive can be implemented straightforwardly in terms of the sched_setaffinity() primitive. Of course, a somewhat less "toy" implementation would restore the affinity upon completion rather than just leaving all tasks running on the last CPU, but when I said "toy", I meant **toy**!

So how the heck is this supposed to work???

Remember that it is illegal to block while in an RCU read-side critical section. Therefore, if a given CPU executes a context switch, we know that it must have completed all preceding RCU read-side critical sections. Once **all** CPUs have executed a context switch, then **all** preceding RCU read-side critical sections will have completed.

So, suppose that we remove a data item from its structure and then invoke synchronize_rcu(). Once synchronize_rcu() returns, we are guaranteed that there are no RCU read-side critical sections holding a reference to that data item, so we can safely reclaim it.

Quick Quiz #2:
Give an example where Classic RCU's read-side overhead is **negative**.

Answers to Quick Quiz

Quick Quiz #3:
If it is illegal to block in an RCU read-side critical section, what the heck do you do in CONFIG_PREEMPT_RT, where normal spinlocks can block???

Answers to Quick Quiz

6. ANALOGY WITH READER-WRITER LOCKING

Although RCU can be used in many different ways, a very common use of RCU is analogous to reader-writer locking. The following unified diff shows how closely related RCU and reader-writer locking can be.

```
@@ -5,5 +5,5 @@ struct el {
        int data;
        /* Other data fields */
 };
-rwlock_t listmutex;
+spinlock_t listmutex;
 struct el head;

@@ -13,15 +14,15 @@
        struct list_head *lp;
        struct el *p;

-       read_lock(&listmutex);
-       list_for_each_entry(p, head, lp) {
+       rcu_read_lock();
+       list_for_each_entry_rcu(p, head, lp) {
                if (p->key == key) {
                        *result = p->data;
-                       read_unlock(&listmutex);
+                       rcu_read_unlock();
                        return 1;
                }
        }
-       read_unlock(&listmutex);
+       rcu_read_unlock();
        return 0;
 }

@@ -29,15 +30,16 @@
 {
        struct el *p;

-       write_lock(&listmutex);
+       spin_lock(&listmutex);
        list_for_each_entry(p, head, lp) {
                if (p->key == key) {
-                       list_del(&p->list);
-                       write_unlock(&listmutex);
+                       list_del_rcu(&p->list);
+                       spin_unlock(&listmutex);
+                       synchronize_rcu();
                        kfree(p);
                        return 1;
                }
        }
```

```
-       write_unlock(&listmutex);
+       spin_unlock(&listmutex);
        return 0;
 }
```

Or, for those who prefer a side-by-side listing:

```
struct el {
  struct list_head list;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
rwlock_t listmutex;
struct el head;
```

```
struct el {
  struct list_head list;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
spinlock_t listmutex;
struct el head;
```

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  read_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listmutex);
      return 1;
    }
  }
  read_unlock(&listmutex);
  return 0;
}
```

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

```
int delete(long key)
{
  struct el *p;

  write_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del(&p->list);
      write_unlock(&listmutex);
      kfree(p);
      return 1;
    }
  }
  write_unlock(&listmutex);
```

```
int delete(long key)
{
  struct el *p;

  spin_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del_rcu(&p->list);
      spin_unlock(&listmutex);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listmutex);
```

```
  return 0;                              return 0;
}                                      }
```

Either way, the differences are quite small. Read-side locking moves to rcu_read_lock() and rcu_read_unlock, update-side locking moves from a reader-writer lock to a simple spinlock, and a synchronize_rcu() precedes the kfree().

However, there is one potential catch: the read-side and update-side critical sections can now run concurrently. In many cases, this will not be a problem, but it is necessary to check carefully regardless. For example, if multiple independent list updates must be seen as a single atomic update, converting to RCU will require special care.

Also, the presence of synchronize_rcu() means that the RCU version of delete() can now block. If this is a problem, there is a callback-based mechanism that never blocks, namely call_rcu() or kfree_rcu(), that can be used in place of synchronize_rcu().

7. ANALOGY WITH REFERENCE COUNTING

The reader-writer analogy (illustrated by the previous section) is not always the best way to think about using RCU. Another helpful analogy considers RCU an effective reference count on everything which is protected by RCU.

A reference count typically does not prevent the referenced object's values from changing, but does prevent changes to type -- particularly the gross change of type that happens when that object's memory is freed and re-allocated for some other purpose. Once a type-safe reference to the object is obtained, some other mechanism is needed to ensure consistent access to the data in the object. This could involve taking a spinlock, but with RCU the typical approach is to perform reads with SMP-aware operations such as smp_load_acquire(), to perform updates with atomic read-modify-write operations, and to provide the necessary ordering. RCU provides a number of support functions that embed the required operations and ordering, such as the list_for_each_entry_rcu() macro used in the previous section.

A more focused view of the reference counting behavior is that, between rcu_read_lock() and rcu_read_unlock(), any reference taken with rcu_dereference() on a pointer marked as `__rcu` can be treated as though a reference-count on that object has been temporarily increased. This prevents the object from changing type. Exactly what this means will depend on normal expectations of objects of that type, but it typically includes that spinlocks can still be safely locked, normal reference counters can be safely manipulated, and `__rcu` pointers can be safely dereferenced.

Some operations that one might expect to see on an object for which an RCU reference is held include:

- Copying out data that is guaranteed to be stable by the object's type.
- Using kref_get_unless_zero() or similar to get a longer-term reference. This may fail of course.
- Acquiring a spinlock in the object, and checking if the object still is the expected object and if so, manipulating it freely.

The understanding that RCU provides a reference that only prevents a change of type is particularly visible with objects allocated from a slab cache marked `SLAB_TYPESAFE_BY_RCU`. RCU operations may yield a reference to an object from such a cache that has been concurrently freed and the memory reallocated to a completely different object, though of the same type. In this case RCU doesn't even protect the identity of the object from changing, only its type. So the object found may not be the one expected, but it will be one where it is safe to take a reference (and then potentially acquiring a spinlock), allowing subsequent code to check whether the identity matches expectations. It is tempting to simply acquire the spinlock without first taking the reference, but unfortunately any spinlock in a `SLAB_TYPESAFE_BY_RCU` object must be initialized after each and every call to kmem_cache_alloc(), which renders reference-free spinlock acquisition completely unsafe. Therefore, when using `SLAB_TYPESAFE_BY_RCU`, make proper use of a reference counter. (Those willing to use a kmem_cache constructor may also use locking, including cache-friendly sequence locking.)

With traditional reference counting -- such as that implemented by the kref library in Linux -- there is typically code that runs when the last reference to an object is dropped. With kref, this is the function passed to kref_put(). When RCU is being used, such finalization code must not be run until all `__rcu` pointers referencing the object have been updated, and then a grace period has passed. Every remaining globally visible pointer to the object must be considered to be a potential counted reference, and the finalization code is typically run using call_rcu() only after all those pointers have been changed.

To see how to choose between these two analogies -- of RCU as a reader-writer lock and RCU as a reference counting system -- it is useful to reflect on the scale of the thing being protected. The reader-writer lock analogy looks at larger multi-part objects such as a linked list and shows how RCU can facilitate concurrency while elements are added to, and removed from, the list. The reference-count analogy looks at the individual objects and looks at how they can be accessed safely within whatever whole they are a part of.

8. FULL LIST OF RCU APIs

The RCU APIs are documented in docbook-format header comments in the Linux-kernel source code, but it helps to have a full list of the APIs, since there does not appear to be a way to categorize them in docbook. Here is the list, by category.

RCU list traversal:

```
list_entry_rcu
list_entry_lockless
list_first_entry_rcu
list_next_rcu
list_for_each_entry_rcu
list_for_each_entry_continue_rcu
list_for_each_entry_from_rcu
list_first_or_null_rcu
list_next_or_null_rcu
hlist_first_rcu
hlist_next_rcu
hlist_pprev_rcu
hlist_for_each_entry_rcu
hlist_for_each_entry_rcu_bh
hlist_for_each_entry_from_rcu
hlist_for_each_entry_continue_rcu
hlist_for_each_entry_continue_rcu_bh
hlist_nulls_first_rcu
hlist_nulls_for_each_entry_rcu
```

```
hlist_bl_first_rcu
hlist_bl_for_each_entry_rcu
```

RCU pointer/list update:

```
rcu_assign_pointer
list_add_rcu
list_add_tail_rcu
list_del_rcu
list_replace_rcu
hlist_add_behind_rcu
hlist_add_before_rcu
hlist_add_head_rcu
hlist_add_tail_rcu
hlist_del_rcu
hlist_del_init_rcu
hlist_replace_rcu
list_splice_init_rcu
list_splice_tail_init_rcu
hlist_nulls_del_init_rcu
hlist_nulls_del_rcu
hlist_nulls_add_head_rcu
hlist_bl_add_head_rcu
hlist_bl_del_init_rcu
hlist_bl_del_rcu
hlist_bl_set_first_rcu
```

RCU:

```
Critical sections       Grace period            Barrier

rcu_read_lock           synchronize_net         rcu_barrier
rcu_read_unlock         synchronize_rcu
rcu_dereference         synchronize_rcu_expedited
rcu_read_lock_held      call_rcu
rcu_dereference_check   kfree_rcu
rcu_dereference_protected
```

bh:

```
Critical sections       Grace period            Barrier

rcu_read_lock_bh        call_rcu                rcu_barrier
rcu_read_unlock_bh      synchronize_rcu
[local_bh_disable]      synchronize_rcu_expedited
[and friends]
rcu_dereference_bh
rcu_dereference_bh_check
rcu_dereference_bh_protected
rcu_read_lock_bh_held
```

sched:

```
Critical sections       Grace period            Barrier

rcu_read_lock_sched     call_rcu                rcu_barrier
rcu_read_unlock_sched   synchronize_rcu
[preempt_disable]       synchronize_rcu_expedited
[and friends]
rcu_read_lock_sched_notrace
rcu_read_unlock_sched_notrace
rcu_dereference_sched
rcu_dereference_sched_check
rcu_dereference_sched_protected
rcu_read_lock_sched_held
```

RCU-Tasks:

```
Critical sections       Grace period            Barrier

N/A                     call_rcu_tasks          rcu_barrier_tasks
                        synchronize_rcu_tasks
```

RCU-Tasks-Rude:

```
Critical sections       Grace period            Barrier

N/A                     call_rcu_tasks_rude     rcu_barrier_tasks_rude
                        synchronize_rcu_tasks_rude
```

RCU-Tasks-Trace:

```
Critical sections       Grace period            Barrier

rcu_read_lock_trace     call_rcu_tasks_trace    rcu_barrier_tasks_trace
rcu_read_unlock_trace   synchronize_rcu_tasks_trace
```

SRCU:

```
Critical sections       Grace period            Barrier

srcu_read_lock          call_srcu               srcu_barrier
srcu_read_unlock        synchronize_srcu
srcu_dereference        synchronize_srcu_expedited
srcu_dereference_check
srcu_read_lock_held
```

SRCU: Initialization/cleanup:

```
DEFINE_SRCU
DEFINE_STATIC_SRCU
init_srcu_struct
cleanup_srcu_struct
```

All: lockdep-checked RCU utility APIs:

```
RCU_LOCKDEP_WARN
rcu_sleep_check
```

All: Unchecked RCU-protected pointer access:

```
rcu_dereference_raw
```

All: Unchecked RCU-protected pointer access with dereferencing prohibited:

```
rcu_access_pointer
```

See the comment headers in the source code (or the docbook generated from them) for more information.

However, given that there are no fewer than four families of RCU APIs in the Linux kernel, how do you choose which one to use? The following list can be helpful:

a. Will readers need to block? If so, you need SRCU.

b. Will readers need to block and are you doing tracing, for example, ftrace or BPF? If so, you need RCU-tasks, RCU-tasks-rude, and/or RCU-tasks-trace.

c. What about the -rt patchset? If readers would need to block in an non-rt kernel, you need SRCU. If readers would block when acquiring spinlocks in a -rt kernel, but not in a non-rt kernel, SRCU is not necessary. (The -rt patchset turns spinlocks into sleeplocks, hence this distinction.)

d. Do you need to treat NMI handlers, hardirq handlers, and code segments with preemption disabled (whether via preempt_disable(), local_irq_save(), local_bh_disable(), or some other mechanism) as if they were explicit RCU readers? If so, RCU-sched readers are the only choice that will work for you, but since about v4.20 you use can use the vanilla RCU update primitives.

e. Do you need RCU grace periods to complete even in the face of softirq monopolization of one or more of the CPUs? For example, is your code subject to network-based denial-of-service attacks? If so, you should disable softirq across your readers, for example, by using *rcu_read_lock_bh()*. Since about v4.20 you use can use the vanilla RCU update primitives.

f. Is your workload too update-intensive for normal use of RCU, but inappropriate for other synchronization mechanisms? If so, consider SLAB_TYPESAFE_BY_RCU (which was originally named SLAB_DESTROY_BY_RCU). But please be careful!

g. Do you need read-side critical sections that are respected even on CPUs that are deep in the idle loop, during entry to or exit from user-mode execution, or on an offlined CPU? If so, SRCU and RCU Tasks Trace are the only choices that will work for you, with SRCU being strongly preferred in almost all cases.

h. Otherwise, use RCU.

Of course, this all assumes that you have determined that RCU is in fact the right tool for your job.

9. ANSWERS TO QUICK QUIZZES

Quick Quiz #1:

Why is this argument naive? How could a deadlock occur when using this algorithm in a real-world Linux kernel? [Referring to the lock-based "toy" RCU algorithm.]

Answer:

Consider the following sequence of events:

1. CPU 0 acquires some unrelated lock, call it "problematic_lock", disabling irq via spin_lock_irqsave().
2. CPU 1 enters synchronize_rcu(), write-acquiring rcu_gp_mutex.
3. CPU 0 enters rcu_read_lock(), but must wait because CPU 1 holds rcu_gp_mutex.
4. CPU 1 is interrupted, and the irq handler attempts to acquire problematic_lock.

The system is now deadlocked.

One way to avoid this deadlock is to use an approach like that of CONFIG_PREEMPT_RT, where all normal spinlocks become blocking locks, and all irq handlers execute in the context of special tasks. In this case, in step 4 above, the irq handler would block, allowing CPU 1 to release rcu_gp_mutex, avoiding the deadlock.

Even in the absence of deadlock, this RCU implementation allows latency to "bleed" from readers to other readers through synchronize_rcu(). To see this, consider task A in an RCU read-side critical section (thus read-holding rcu_gp_mutex), task B blocked attempting to write-acquire rcu_gp_mutex, and task C blocked in rcu_read_lock() attempting to read_acquire rcu_gp_mutex. Task A's RCU read-side latency is holding up task C, albeit indirectly via task B.

Realtime RCU implementations therefore use a counter-based approach where tasks in RCU read-side critical sections cannot be blocked by tasks executing synchronize_rcu().

Back to Quick Quiz #1

Quick Quiz #2:

Give an example where Classic RCU's read-side overhead is **negative**.

Answer:

Imagine a single-CPU system with a non-CONFIG_PREEMPTION kernel where a routing table is used by process-context code, but can be updated by irq-context code (for example, by an "ICMP REDIRECT" packet). The usual way of handling this would be to have the process-context code disable interrupts while searching the routing table. Use of RCU allows such interrupt-disabling to be dispensed with. Thus, without RCU, you pay the cost of disabling interrupts, and with RCU you don't.

One can argue that the overhead of RCU in this case is negative with respect to the single-CPU interrupt-disabling approach. Others might argue that the overhead of RCU is merely zero, and that replacing the positive overhead of the interrupt-disabling scheme with the zero-overhead RCU scheme does not constitute negative overhead.

In real life, of course, things are more complex. But even the theoretical possibility of negative overhead for a synchronization primitive is a bit unexpected. ;-)

Back to Quick Quiz #2

Quick Quiz #3:

If it is illegal to block in an RCU read-side critical section, what the heck do you do in CONFIG_PREEMPT_RT, where normal spinlocks can block???

Answer:

Just as CONFIG_PREEMPT_RT permits preemption of spinlock critical sections, it permits preemption of RCU read-side critical sections. It also permits spinlocks blocking while in RCU read-side critical sections.

Why the apparent inconsistency? Because it is possible to use priority boosting to keep the RCU grace periods short if need be (for example, if running short of memory). In contrast, if blocking waiting for (say) network reception, there is no way to know what should be boosted. Especially given that the process we need to boost might well be a human being who just went out for a pizza or something. And although a computer-operated cattle prod might arouse serious interest, it might also provoke serious objections. Besides, how does the computer know what pizza parlor the human being went to???

Back to Quick Quiz #3

ACKNOWLEDGEMENTS

My thanks to the people who helped make this human-readable, including Jon Walpole, Josh Triplett, Serge Hallyn, Suzanne Wood, and Alan Stern.

For more information, see http://www.rdrop.com/users/paulmck/RCU.

4.5.7 RCU Concepts

The basic idea behind RCU (read-copy update) is to split destructive operations into two parts, one that prevents anyone from seeing the data item being destroyed, and one that actually carries out the destruction. A "grace period" must elapse between the two parts, and this grace period must be long enough that any readers accessing the item being deleted have since dropped their references. For example, an RCU-protected deletion from a linked list would first remove the item from the list, wait for a grace period to elapse, then free the element. See listRCU.rst for more information on using RCU with linked lists.

Frequently Asked Questions

- Why would anyone want to use RCU?

  The advantage of RCU's two-part approach is that RCU readers need not acquire any locks, perform any atomic instructions, write to shared memory, or (on CPUs other than Alpha) execute any memory barriers. The fact that these operations are quite expensive on modern CPUs is what gives RCU its performance advantages in read-mostly situations. The fact that RCU readers need not acquire locks can also greatly simplify deadlock-avoidance code.

- How can the updater tell when a grace period has completed if the RCU readers give no indication when they are done?

  Just as with spinlocks, RCU readers are not permitted to block, switch to user-mode execution, or enter the idle loop. Therefore, as soon as a CPU is seen passing through any of these three states, we know that that CPU has exited any previous RCU read-side critical sections. So, if we remove an item from a linked list, and then wait until all CPUs have switched context, executed in user mode, or executed in the idle loop, we can safely free up that item.

Preemptible variants of RCU (CONFIG_PREEMPT_RCU) get the same effect, but require that the readers manipulate CPU-local counters. These counters allow limited types of blocking within RCU read-side critical sections. SRCU also uses CPU-local counters, and permits general blocking within RCU read-side critical sections. These variants of RCU detect grace periods by sampling these counters.

- If I am running on a uniprocessor kernel, which can only do one thing at a time, why should I wait for a grace period?

  See UP.rst for more information.

- How can I see where RCU is currently used in the Linux kernel?

  Search for "rcu_read_lock", "rcu_read_unlock", "call_rcu", "rcu_read_lock_bh", "rcu_read_unlock_bh", "srcu_read_lock", "srcu_read_unlock", "synchronize_rcu", "synchronize_net", "synchronize_srcu", and the other RCU primitives. Or grab one of the cscope databases from:

  (http://www.rdrop.com/users/paulmck/RCU/linuxusage/rculocktab.html).

- What guidelines should I follow when writing code that uses RCU?

  See checklist.rst.

- Why the name "RCU"?

  "RCU" stands for "read-copy update". listRCU.rst has more information on where this name came from, search for "read-copy update" to find it.

- I hear that RCU is patented? What is with that?

  Yes, it is. There are several known patents related to RCU, search for the string "Patent" in Documentation/RCU/RTFP.txt to find them. Of these, one was allowed to lapse by the assignee, and the others have been contributed to the Linux kernel under GPL. Many (but not all) have long since expired. There are now also LGPL implementations of user-level RCU available (https://liburcu.org/).

- I hear that RCU needs work in order to support realtime kernels?

  Realtime-friendly RCU are enabled via the CONFIG_PREEMPTION kernel configuration parameter.

- Where can I find more information on RCU?

  See the Documentation/RCU/RTFP.txt file. Or point your browser at (https://docs.google.com/document/d/1X0lThx8OK0ZgLMqVoXiR4ZrGURHrXK6NyLRbeXe3Xac/edit) or (https://docs.google.com/document/d/1GCdQC8SDbb54W1shjEXqGZ0Rq8a6kIeYutdSIajfpLA/edit?usp=sharing).

4.5.8 Using RCU hlist_nulls to protect list and objects

This section describes how to use hlist_nulls to protect read-mostly linked lists and objects using SLAB_TYPESAFE_BY_RCU allocations.

Please read the basics in listRCU.rst.

Using 'nulls'

Using special makers (called 'nulls') is a convenient way to solve following problem.

Without 'nulls', a typical RCU linked list managing objects which are allocated with SLAB_TYPESAFE_BY_RCU kmem_cache can use the following algorithms. Following examples assume 'obj' is a pointer to such objects, which is having below type.

```
struct object {
  struct hlist_node obj_node;
  atomic_t refcnt;
  unsigned int key;
};
```

1) Lookup algorithm

```
begin:
rcu_read_lock();
obj = lockless_lookup(key);
if (obj) {
  if (!try_get_ref(obj)) { // might fail for free objects
    rcu_read_unlock();
    goto begin;
  }
  /*
  * Because a writer could delete object, and a writer could
  * reuse these object before the RCU grace period, we
  * must check key after getting the reference on object
  */
  if (obj->key != key) { // not the object we expected
    put_ref(obj);
    rcu_read_unlock();
    goto begin;
  }
}
rcu_read_unlock();
```

Beware that lockless_lookup(key) cannot use traditional *hlist_for_each_entry_rcu()* but a version with an additional memory barrier (smp_rmb())

```
lockless_lookup(key)
{
  struct hlist_node *node, *next;
```

```
  for (pos = rcu_dereference((head)->first);
       pos && ({ next = pos->next; smp_rmb(); prefetch(next); 1; }) &&
       ({ obj = hlist_entry(pos, typeof(*obj), obj_node); 1; });
       pos = rcu_dereference(next))
    if (obj->key == key)
      return obj;
  return NULL;
}
```

And note the traditional *hlist_for_each_entry_rcu()* misses this smp_rmb():

```
struct hlist_node *node;
for (pos = rcu_dereference((head)->first);
     pos && ({ prefetch(pos->next); 1; }) &&
     ({ obj = hlist_entry(pos, typeof(*obj), obj_node); 1; });
     pos = rcu_dereference(pos->next))
  if (obj->key == key)
    return obj;
return NULL;
```

Quoting Corey Minyard:

```
"If the object is moved from one list to another list in-between the
time the hash is calculated and the next field is accessed, and the
object has moved to the end of a new list, the traversal will not
complete properly on the list it should have, since the object will
be on the end of the new list and there's not a way to tell it's on a
new list and restart the list traversal. I think that this can be
solved by pre-fetching the "next" field (with proper barriers) before
checking the key."
```

2) Insertion algorithm

We need to make sure a reader cannot read the new 'obj->obj_node.next' value and previous value of 'obj->key'. Otherwise, an item could be deleted from a chain, and inserted into another chain. If new chain was empty before the move, 'next' pointer is NULL, and lockless reader can not detect the fact that it missed following items in original chain.

```
/*
 * Please note that new inserts are done at the head of list,
 * not in the middle or end.
 */
obj = kmem_cache_alloc(...);
lock_chain(); // typically a spin_lock()
obj->key = key;
atomic_set_release(&obj->refcnt, 1); // key before refcnt
hlist_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```

3) Removal algorithm

Nothing special here, we can use a standard RCU hlist deletion. But thanks to SLAB_TYPESAFE_BY_RCU, beware a deleted object can be reused very very fast (before the end of RCU grace period)

```
if (put_last_reference_on(obj) {
  lock_chain(); // typically a spin_lock()
  hlist_del_init_rcu(&obj->obj_node);
  unlock_chain(); // typically a spin_unlock()
  kmem_cache_free(cachep, obj);
}
```

Avoiding extra smp_rmb()

With hlist_nulls we can avoid extra smp_rmb() in lockless_lookup().

For example, if we choose to store the slot number as the 'nulls' end-of-list marker for each slot of the hash table, we can detect a race (some writer did a delete and/or a move of an object to another chain) checking the final 'nulls' value if the lookup met the end of chain. If final 'nulls' value is not the slot number, then we must restart the lookup at the beginning. If the object was moved to the same chain, then the reader doesn't care: It might occasionally scan the list again without harm.

Note that using hlist_nulls means the type of 'obj_node' field of 'struct object' becomes 'struct hlist_nulls_node'.

1) lookup algorithm

```
head = &table[slot];
begin:
rcu_read_lock();
hlist_nulls_for_each_entry_rcu(obj, node, head, obj_node) {
  if (obj->key == key) {
    if (!try_get_ref(obj)) { // might fail for free objects
      rcu_read_unlock();
      goto begin;
    }
    if (obj->key != key) { // not the object we expected
      put_ref(obj);
      rcu_read_unlock();
      goto begin;
    }
    goto out;
  }
}

// If the nulls value we got at the end of this lookup is
```

```
// not the expected one, we must restart lookup.
// We probably met an item that was moved to another chain.
if (get_nulls_value(node) != slot) {
  put_ref(obj);
  rcu_read_unlock();
  goto begin;
}
obj = NULL;

out:
rcu_read_unlock();
```

2) Insert algorithm

Same to the above one, but uses *hlist_nulls_add_head_rcu()* instead of hlist_add_head_rcu().

```
/*
 * Please note that new inserts are done at the head of list,
 * not in the middle or end.
 */
obj = kmem_cache_alloc(cachep);
lock_chain(); // typically a spin_lock()
obj->key = key;
atomic_set_release(&obj->refcnt, 1); // key before refcnt
/*
 * insert obj in RCU way (readers might be traversing chain)
 */
hlist_nulls_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```

4.5.9 Reference-count design for elements of lists/arrays protected by RCU

Please note that the percpu-ref feature is likely your first stop if you need to combine reference counts and RCU. Please see include/linux/percpu-refcount.h for more information. However, in those unusual cases where percpu-ref would consume too much memory, please read on.

Reference counting on elements of lists which are protected by traditional reader/writer spinlocks or semaphores are straightforward:

CODE LISTING A:

```
1.                                      2.
add()                                   search_and_reference()
{                                       {
    alloc_object                            read_lock(&list_lock);
    ...                                     search_for_element
    atomic_set(&el->rc, 1);                 atomic_inc(&el->rc);
    write_lock(&list_lock);                  ...
```

```
    add_element                                 read_unlock(&list_lock);
    ...                                         ...
    write_unlock(&list_lock);               }
}

3.                                          4.
release_referenced()                        delete()
{                                           {
    ...                                         write_lock(&list_lock);
    if(atomic_dec_and_test(&el->rc))            ...
        kfree(el);
    ...                                         remove_element
}                                               write_unlock(&list_lock);
                                                ...
                                                if (atomic_dec_and_test(&el->rc))
                                                    kfree(el);
                                                ...
                                            }
```

If this list/array is made lock free using RCU as in changing the write_lock() in add() and delete() to spin_lock() and changing read_lock() in search_and_reference() to rcu_read_lock(), the atomic_inc() in search_and_reference() could potentially hold reference to an element which has already been deleted from the list/array. Use atomic_inc_not_zero() in this scenario as follows:

CODE LISTING B:

```
1.                                          2.
add()                                       search_and_reference()
{                                           {
    alloc_object                                rcu_read_lock();
    ...                                         search_for_element
    atomic_set(&el->rc, 1);                     if (!atomic_inc_not_zero(&el->rc))
 ↪{
    spin_lock(&list_lock);                          rcu_read_unlock();
                                                    return FAIL;
    add_element                                 }
    ...                                         ...
    spin_unlock(&list_lock);                    rcu_read_unlock();
}                                           }
3.                                          4.
release_referenced()                        delete()
{                                           {
    ...                                         spin_lock(&list_lock);
    if (atomic_dec_and_test(&el->rc))           ...
        call_rcu(&el->head, el_free);           remove_element
    ...                                         spin_unlock(&list_lock);
}                                               ...
                                                if (atomic_dec_and_test(&el->rc))
                                                    call_rcu(&el->head, el_free);
                                                ...
```

```
                                        }
```

Sometimes, a reference to the element needs to be obtained in the update (write) stream. In such cases, atomic_inc_not_zero() might be overkill, since we hold the update-side spinlock. One might instead use atomic_inc() in such cases.

It is not always convenient to deal with "FAIL" in the search_and_reference() code path. In such cases, the atomic_dec_and_test() may be moved from delete() to el_free() as follows:

CODE LISTING C:

```
1.
add()
{
    alloc_object
    ...
    atomic_set(&el->rc, 1);
    spin_lock(&list_lock);

    add_element
    ...
    spin_unlock(&list_lock);
}

2.
search_and_reference()
{
    rcu_read_lock();
    search_for_element
    atomic_inc(&el->rc);
    ...

    rcu_read_unlock();
}

3.
release_referenced()
{
    ...
    if (atomic_dec_and_test(&el->rc))
        kfree(el);
    ...
}

4.
delete()
{
    spin_lock(&list_lock);
    ...
    remove_element
    spin_unlock(&list_lock);
    ...
    call_rcu(&el->head, el_free);
    ...
}

5.
void el_free(struct rcu_head *rhp)
{
    release_referenced();
}
```

The key point is that the initial reference added by add() is not removed until after a grace period has elapsed following removal. This means that search_and_reference() cannot find this element, which means that the value of el->rc cannot increase. Thus, once it reaches zero, there are no readers that can or ever will be able to reference the element. The element can therefore safely be freed. This in turn guarantees that if any reader finds the element, that reader may safely acquire a reference without checking the value of the reference counter.

A clear advantage of the RCU-based pattern in listing C over the one in listing B is that any call to search_and_reference() that locates a given object will succeed in obtaining a reference to that object, even given a concurrent invocation of delete() for that same object. Similarly, a clear advantage of both listings B and C over listing A is that a call to delete() is not delayed even if there are an arbitrarily large number of calls to search_and_reference() searching for the same object that delete() was invoked on. Instead, all that is delayed is the eventual invocation of kfree(), which is usually not a problem on modern computer systems, even the small ones.

In cases where delete() can sleep, synchronize_rcu() can be called from delete(), so that el_free()

can be subsumed into delete as follows:

```
4.
delete()
{
    spin_lock(&list_lock);
    ...
    remove_element
    spin_unlock(&list_lock);
    ...
    synchronize_rcu();
    if (atomic_dec_and_test(&el->rc))
        kfree(el);
    ...
}
```

As additional examples in the kernel, the pattern in listing C is used by reference counting of struct pid, while the pattern in listing B is used by struct posix_acl.

4.5.10 RCU Torture Test Operation

CONFIG_RCU_TORTURE_TEST

The CONFIG_RCU_TORTURE_TEST config option is available for all RCU implementations. It creates an rcutorture kernel module that can be loaded to run a torture test. The test periodically outputs status messages via printk(), which can be examined via the dmesg command (perhaps grepping for "torture"). The test is started when the module is loaded, and stops when the module is unloaded.

Module parameters are prefixed by "rcutorture." in Documentation/admin-guide/kernel-parameters.txt.

Output

The statistics output is as follows:

```
rcu-torture:--- Start of test: nreaders=16 nfakewriters=4 stat_interval=30␣
↪verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=5 irqreader=1 fqs_
↪duration=0 fqs_holdoff=0 fqs_stutter=3 test_boost=1/0 test_boost_interval=7␣
↪test_boost_duration=4
rcu-torture: rtc:           (null) ver: 155441 tfle: 0 rta: 155441 rtaf: 8884␣
↪rtf: 155440 rtmbe: 0 rtbe: 0 rtbke: 0 rtbre: 0 rtbf: 0 rtb: 0 nt: 3055767
rcu-torture: Reader Pipe:  727860534 34213 0 0 0 0 0 0 0 0 0
rcu-torture: Reader Batch:  727877838 17003 0 0 0 0 0 0 0 0 0
rcu-torture: Free-Block Circulation:  155440 155440 155440 155440 155440␣
↪155440 155440 155440 155440 155440 0
rcu-torture:--- End of test: SUCCESS: nreaders=16 nfakewriters=4 stat_
↪interval=30 verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=5␣
↪irqreader=1 fqs_duration=0 fqs_holdoff=0 fqs_stutter=3 test_boost=1/0 test_
↪boost_interval=7 test_boost_duration=4
```

The command "dmesg | grep torture:" will extract this information on most systems. On more esoteric configurations, it may be necessary to use other commands to access the output of the printk()s used by the RCU torture test. The printk()s use KERN_ALERT, so they should be evident. ;-)

The first and last lines show the rcutorture module parameters, and the last line shows either "SUCCESS" or "FAILURE", based on rcutorture's automatic determination as to whether RCU operated correctly.

The entries are as follows:

- "rtc": The hexadecimal address of the structure currently visible to readers.
- "ver": The number of times since boot that the RCU writer task has changed the structure visible to readers.
- "tfle": If non-zero, indicates that the "torture freelist" containing structures to be placed into the "rtc" area is empty. This condition is important, since it can fool you into thinking that RCU is working when it is not. :-/
- "rta": Number of structures allocated from the torture freelist.
- "rtaf": Number of allocations from the torture freelist that have failed due to the list being empty. It is not unusual for this to be non-zero, but it is bad for it to be a large fraction of the value indicated by "rta".
- "rtf": Number of frees into the torture freelist.
- "rtmbe": A non-zero value indicates that rcutorture believes that rcu_assign_pointer() and rcu_dereference() are not working correctly. This value should be zero.
- "rtbe": A non-zero value indicates that one of the `rcu_barrier()` family of functions is not working correctly.
- "rtbke": rcutorture was unable to create the real-time kthreads used to force RCU priority inversion. This value should be zero.
- "rtbre": Although rcutorture successfully created the kthreads used to force RCU priority inversion, it was unable to set them to the real-time priority level of 1. This value should be zero.
- "rtbf": The number of times that RCU priority boosting failed to resolve RCU priority inversion.
- "rtb": The number of times that rcutorture attempted to force an RCU priority inversion condition. If you are testing RCU priority boosting via the "test_boost" module parameter, this value should be non-zero.
- "nt": The number of times rcutorture ran RCU read-side code from within a timer handler. This value should be non-zero only if you specified the "irqreader" module parameter.
- "Reader Pipe": Histogram of "ages" of structures seen by readers. If any entries past the first two are non-zero, RCU is broken. And rcutorture prints the error flag string "!!!" to make sure you notice. The age of a newly allocated structure is zero, it becomes one when removed from reader visibility, and is incremented once per grace period subsequently -- and is freed after passing through (RCU_TORTURE_PIPE_LEN-2) grace periods.

  The output displayed above was taken from a correctly working RCU. If you want to see what it looks like when broken, break it yourself. ;-)

- "Reader Batch": Another histogram of "ages" of structures seen by readers, but in terms of counter flips (or batches) rather than in terms of grace periods. The legal number of non-zero entries is again two. The reason for this separate view is that it is sometimes easier to get the third entry to show up in the "Reader Batch" list than in the "Reader Pipe" list.
- "Free-Block Circulation": Shows the number of torture structures that have reached a given point in the pipeline. The first element should closely correspond to the number of structures allocated, the second to the number that have been removed from reader view, and all but the last remaining to the corresponding number of passes through a grace period. The last entry should be zero, as it is only incremented if a torture structure's counter somehow gets incremented farther than it should.

Different implementations of RCU can provide implementation-specific additional information. For example, Tree SRCU provides the following additional line:

```
srcud-torture: Tree SRCU per-CPU(idx=0): 0(35,-21) 1(-4,24) 2(1,1) 3(-26,20)␣
↪4(28,-47) 5(-9,4) 6(-10,14) 7(-14,11) T(1,6)
```

This line shows the per-CPU counter state, in this case for Tree SRCU using a dynamically allocated srcu_struct (hence "srcud-" rather than "srcu-"). The numbers in parentheses are the values of the "old" and "current" counters for the corresponding CPU. The "idx" value maps the "old" and "current" values to the underlying array, and is useful for debugging. The final "T" entry contains the totals of the counters.

Usage on Specific Kernel Builds

It is sometimes desirable to torture RCU on a specific kernel build, for example, when preparing to put that kernel build into production. In that case, the kernel should be built with CONFIG_RCU_TORTURE_TEST=m so that the test can be started using modprobe and terminated using rmmod.

For example, the following script may be used to torture RCU:

```
#!/bin/sh

modprobe rcutorture
sleep 3600
rmmod rcutorture
dmesg | grep torture:
```

The output can be manually inspected for the error flag of "!!!". One could of course create a more elaborate script that automatically checked for such errors. The "rmmod" command forces a "SUCCESS", "FAILURE", or "RCU_HOTPLUG" indication to be printk()ed. The first two are self-explanatory, while the last indicates that while there were no RCU failures, CPU-hotplug problems were detected.

Usage on Mainline Kernels

When using rcutorture to test changes to RCU itself, it is often necessary to build a number of kernels in order to test that change across a broad range of combinations of the relevant Kconfig options and of the relevant kernel boot parameters. In this situation, use of modprobe and rmmod can be quite time-consuming and error-prone.

Therefore, the tools/testing/selftests/rcutorture/bin/kvm.sh script is available for mainline testing for x86, arm64, and powerpc. By default, it will run the series of tests specified by tools/testing/selftests/rcutorture/configs/rcu/CFLIST, with each test running for 30 minutes within a guest OS using a minimal userspace supplied by an automatically generated initrd. After the tests are complete, the resulting build products and console output are analyzed for errors and the results of the runs are summarized.

On larger systems, rcutorture testing can be accelerated by passing the --cpus argument to kvm.sh. For example, on a 64-CPU system, "--cpus 43" would use up to 43 CPUs to run tests concurrently, which as of v5.4 would complete all the scenarios in two batches, reducing the time to complete from about eight hours to about one hour (not counting the time to build the sixteen kernels). The "--dryrun sched" argument will not run tests, but rather tell you how the tests would be scheduled into batches. This can be useful when working out how many CPUs to specify in the --cpus argument.

Not all changes require that all scenarios be run. For example, a change to Tree SRCU might run only the SRCU-N and SRCU-P scenarios using the --configs argument to kvm.sh as follows: "--configs 'SRCU-N SRCU-P'". Large systems can run multiple copies of of the full set of scenarios, for example, a system with 448 hardware threads can run five instances of the full set concurrently. To make this happen:

```
kvm.sh --cpus 448 --configs '5*CFLIST'
```

Alternatively, such a system can run 56 concurrent instances of a single eight-CPU scenario:

```
kvm.sh --cpus 448 --configs '56*TREE04'
```

Or 28 concurrent instances of each of two eight-CPU scenarios:

```
kvm.sh --cpus 448 --configs '28*TREE03 28*TREE04'
```

Of course, each concurrent instance will use memory, which can be limited using the --memory argument, which defaults to 512M. Small values for memory may require disabling the callback-flooding tests using the --bootargs parameter discussed below.

Sometimes additional debugging is useful, and in such cases the --kconfig parameter to kvm.sh may be used, for example, `--kconfig 'CONFIG_RCU_EQS_DEBUG=y'`. In addition, there are the --gdb, --kasan, and --kcsan parameters. Note that --gdb limits you to one scenario per kvm.sh run and requires that you have another window open from which to run `gdb` as instructed by the script.

Kernel boot arguments can also be supplied, for example, to control rcutorture's module parameters. For example, to test a change to RCU's CPU stall-warning code, use "--bootargs 'rcutorture.stall_cpu=30'". This will of course result in the scripting reporting a failure, namely the resulting RCU CPU stall warning. As noted above, reducing memory may require disabling rcutorture's callback-flooding tests:

```
kvm.sh --cpus 448 --configs '56*TREE04' --memory 128M \
        --bootargs 'rcutorture.fwd_progress=0'
```

Sometimes all that is needed is a full set of kernel builds. This is what the --buildonly parameter does.

The --duration parameter can override the default run time of 30 minutes. For example, `--duration 2d` would run for two days, `--duration 3h` would run for three hours, `--duration 5m` would run for five minutes, and `--duration 45s` would run for 45 seconds. This last can be useful for tracking down rare boot-time failures.

Finally, the --trust-make parameter allows each kernel build to reuse what it can from the previous kernel build. Please note that without the --trust-make parameter, your tags files may be demolished.

There are additional more arcane arguments that are documented in the source code of the kvm.sh script.

If a run contains failures, the number of buildtime and runtime failures is listed at the end of the kvm.sh output, which you really should redirect to a file. The build products and console output of each run is kept in tools/testing/selftests/rcutorture/res in timestamped directories. A given directory can be supplied to kvm-find-errors.sh in order to have it cycle you through summaries of errors and full error logs. For example:

```
tools/testing/selftests/rcutorture/bin/kvm-find-errors.sh \
        tools/testing/selftests/rcutorture/res/2020.01.20-15.54.23
```

However, it is often more convenient to access the files directly. Files pertaining to all scenarios in a run reside in the top-level directory (2020.01.20-15.54.23 in the example above), while per-scenario files reside in a subdirectory named after the scenario (for example, "TREE04"). If a given scenario ran more than once (as in "--configs '56*TREE04'" above), the directories corresponding to the second and subsequent runs of that scenario include a sequence number, for example, "TREE04.2", "TREE04.3", and so on.

The most frequently used file in the top-level directory is testid.txt. If the test ran in a git repository, then this file contains the commit that was tested and any uncommitted changes in diff format.

The most frequently used files in each per-scenario-run directory are:

.config:
This file contains the Kconfig options.

Make.out:
This contains build output for a specific scenario.

console.log:
This contains the console output for a specific scenario. This file may be examined once the kernel has booted, but it might not exist if the build failed.

vmlinux:
This contains the kernel, which can be useful with tools like objdump and gdb.

A number of additional files are available, but are less frequently used. Many are intended for debugging of rcutorture itself or of its scripting.

As of v5.4, a successful run with the default set of scenarios produces the following summary at the end of the run on a 12-CPU system:

```
SRCU-N ------- 804233 GPs (148.932/s) [srcu: g10008272 f0x0 ]
SRCU-P ------- 202320 GPs (37.4667/s) [srcud: g1809476 f0x0 ]
SRCU-t ------- 1122086 GPs (207.794/s) [srcu: g0 f0x0 ]
SRCU-u ------- 1111285 GPs (205.794/s) [srcud: g1 f0x0 ]
TASKS01 ------- 19666 GPs (3.64185/s) [tasks: g0 f0x0 ]
TASKS02 ------- 20541 GPs (3.80389/s) [tasks: g0 f0x0 ]
TASKS03 ------- 19416 GPs (3.59556/s) [tasks: g0 f0x0 ]
TINY01 ------- 836134 GPs (154.84/s) [rcu: g0 f0x0 ] n_max_cbs: 34198
TINY02 ------- 850371 GPs (157.476/s) [rcu: g0 f0x0 ] n_max_cbs: 2631
TREE01 ------- 162625 GPs (30.1157/s) [rcu: g1124169 f0x0 ]
TREE02 ------- 333003 GPs (61.6672/s) [rcu: g2647753 f0x0 ] n_max_cbs: 35844
TREE03 ------- 306623 GPs (56.782/s) [rcu: g2975325 f0x0 ] n_max_cbs: 1496497
CPU count limited from 16 to 12
TREE04 ------- 246149 GPs (45.5831/s) [rcu: g1695737 f0x0 ] n_max_cbs: 434961
TREE05 ------- 314603 GPs (58.2598/s) [rcu: g2257741 f0x2 ] n_max_cbs: 193997
TREE07 ------- 167347 GPs (30.9902/s) [rcu: g1079021 f0x0 ] n_max_cbs: 478732
CPU count limited from 16 to 12
TREE09 ------- 752238 GPs (139.303/s) [rcu: g13075057 f0x0 ] n_max_cbs: 99011
```

Repeated Runs

Suppose that you are chasing down a rare boot-time failure. Although you could use kvm.sh, doing so will rebuild the kernel on each run. If you need (say) 1,000 runs to have confidence that you have fixed the bug, these pointless rebuilds can become extremely annoying.

This is why kvm-again.sh exists.

Suppose that a previous kvm.sh run left its output in this directory:

```
tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28
```

Then this run can be re-run without rebuilding as follow:

> kvm-again.sh tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28

A few of the original run's kvm.sh parameters may be overridden, perhaps most notably --duration and --bootargs. For example:

```
kvm-again.sh tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28 \
        --duration 45s
```

would re-run the previous test, but for only 45 seconds, thus facilitating tracking down the aforementioned rare boot-time failure.

Distributed Runs

Although kvm.sh is quite useful, its testing is confined to a single system. It is not all that hard to use your favorite framework to cause (say) 5 instances of kvm.sh to run on your 5 systems, but this will very likely unnecessarily rebuild kernels. In addition, manually distributing the desired rcutorture scenarios across the available systems can be painstaking and error-prone.

And this is why the kvm-remote.sh script exists.

If you the following command works:

```
ssh system0 date
```

and if it also works for system1, system2, system3, system4, and system5, and all of these systems have 64 CPUs, you can type:

```
kvm-remote.sh "system0 system1 system2 system3 system4 system5" \
        --cpus 64 --duration 8h --configs "5*CFLIST"
```

This will build each default scenario's kernel on the local system, then spread each of five instances of each scenario over the systems listed, running each scenario for eight hours. At the end of the runs, the results will be gathered, recorded, and printed. Most of the parameters that kvm.sh will accept can be passed to kvm-remote.sh, but the list of systems must come first.

The kvm.sh `--dryrun scenarios` argument is useful for working out how many scenarios may be run in one batch across a group of systems.

You can also re-run a previous remote run in a manner similar to kvm.sh:

kvm-remote.sh "system0 system1 system2 system3 system4 system5"
tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28-remote --duration 24h

In this case, most of the kvm-again.sh parameters may be supplied following the pathname of the old run-results directory.

4.5.11 Using RCU's CPU Stall Detector

This document first discusses what sorts of issues RCU's CPU stall detector can locate, and then discusses kernel parameters and Kconfig options that can be used to fine-tune the detector's operation. Finally, this document explains the stall detector's "splat" format.

What Causes RCU CPU Stall Warnings?

So your kernel printed an RCU CPU stall warning. The next question is "What caused it?" The following problems can result in RCU CPU stall warnings:

- A CPU looping in an RCU read-side critical section.
- A CPU looping with interrupts disabled.
- A CPU looping with preemption disabled.
- A CPU looping with bottom halves disabled.

- For !CONFIG_PREEMPTION kernels, a CPU looping anywhere in the kernel without potentially invoking schedule(). If the looping in the kernel is really expected and desirable behavior, you might need to add some calls to cond_resched().
- Booting Linux using a console connection that is too slow to keep up with the boot-time console-message rate. For example, a 115Kbaud serial console can be *way* too slow to keep up with boot-time message rates, and will frequently result in RCU CPU stall warning messages. Especially if you have added debug printk()s.
- Anything that prevents RCU's grace-period kthreads from running. This can result in the "All QSes seen" console-log message. This message will include information on when the kthread last ran and how often it should be expected to run. It can also result in the `rcu_.*kthread starved for` console-log message, which will include additional debugging information.
- A CPU-bound real-time task in a CONFIG_PREEMPTION kernel, which might happen to preempt a low-priority task in the middle of an RCU read-side critical section. This is especially damaging if that low-priority task is not permitted to run on any other CPU, in which case the next RCU grace period can never complete, which will eventually cause the system to run out of memory and hang. While the system is in the process of running itself out of memory, you might see stall-warning messages.
- A CPU-bound real-time task in a CONFIG_PREEMPT_RT kernel that is running at a higher priority than the RCU softirq threads. This will prevent RCU callbacks from ever being invoked, and in a CONFIG_PREEMPT_RCU kernel will further prevent RCU grace periods from ever completing. Either way, the system will eventually run out of memory and hang. In the CONFIG_PREEMPT_RCU case, you might see stall-warning messages.

  You can use the rcutree.kthread_prio kernel boot parameter to increase the scheduling priority of RCU's kthreads, which can help avoid this problem. However, please note that doing this can increase your system's context-switch rate and thus degrade performance.
- A periodic interrupt whose handler takes longer than the time interval between successive pairs of interrupts. This can prevent RCU's kthreads and softirq handlers from running. Note that certain high-overhead debugging options, for example the function_graph tracer, can result in interrupt handler taking considerably longer than normal, which can in turn result in RCU CPU stall warnings.
- Testing a workload on a fast system, tuning the stall-warning timeout down to just barely avoid RCU CPU stall warnings, and then running the same workload with the same stall-warning timeout on a slow system. Note that thermal throttling and on-demand governors can cause a single system to be sometimes fast and sometimes slow!
- A hardware or software issue shuts off the scheduler-clock interrupt on a CPU that is not in dyntick-idle mode. This problem really has happened, and seems to be most likely to result in RCU CPU stall warnings for CONFIG_NO_HZ_COMMON=n kernels.
- A hardware or software issue that prevents time-based wakeups from occurring. These issues can range from misconfigured or buggy timer hardware through bugs in the interrupt or exception path (whether hardware, firmware, or software) through bugs in Linux's timer subsystem through bugs in the scheduler, and, yes, even including bugs in RCU itself. It can also result in the `rcu_.*timer wakeup didn't happen for` console-log message, which will include additional debugging information.
- A low-level kernel issue that either fails to invoke one of the variants of rcu_eqs_enter(true), rcu_eqs_exit(true), ct_idle_enter(), ct_idle_exit(), ct_irq_enter(), or ct_irq_exit() on the

one hand, or that invokes one of them too many times on the other. Historically, the most frequent issue has been an omission of either irq_enter() or irq_exit(), which in turn invoke ct_irq_enter() or ct_irq_exit(), respectively. Building your kernel with CONFIG_RCU_EQS_DEBUG=y can help track down these types of issues, which sometimes arise in architecture-specific code.

- A bug in the RCU implementation.
- A hardware failure. This is quite unlikely, but is not at all uncommon in large datacenter. In one memorable case some decades back, a CPU failed in a running system, becoming unresponsive, but not causing an immediate crash. This resulted in a series of RCU CPU stall warnings, eventually leading the realization that the CPU had failed.

The RCU, RCU-sched, RCU-tasks, and RCU-tasks-trace implementations have CPU stall warning. Note that SRCU does *not* have CPU stall warnings. Please note that RCU only detects CPU stalls when there is a grace period in progress. No grace period, no CPU stall warnings.

To diagnose the cause of the stall, inspect the stack traces. The offending function will usually be near the top of the stack. If you have a series of stall warnings from a single extended stall, comparing the stack traces can often help determine where the stall is occurring, which will usually be in the function nearest the top of that portion of the stack which remains the same from trace to trace. If you can reliably trigger the stall, ftrace can be quite helpful.

RCU bugs can often be debugged with the help of CONFIG_RCU_TRACE and with RCU's event tracing. For information on RCU's event tracing, see include/trace/events/rcu.h.

Fine-Tuning the RCU CPU Stall Detector

The rcuupdate.rcu_cpu_stall_suppress module parameter disables RCU's CPU stall detector, which detects conditions that unduly delay RCU grace periods. This module parameter enables CPU stall detection by default, but may be overridden via boot-time parameter or at runtime via sysfs. The stall detector's idea of what constitutes "unduly delayed" is controlled by a set of kernel configuration variables and cpp macros:

CONFIG_RCU_CPU_STALL_TIMEOUT

This kernel configuration parameter defines the period of time that RCU will wait from the beginning of a grace period until it issues an RCU CPU stall warning. This time period is normally 21 seconds.

This configuration parameter may be changed at runtime via the /sys/module/rcupdate/parameters/rcu_cpu_stall_timeout, however this parameter is checked only at the beginning of a cycle. So if you are 10 seconds into a 40-second stall, setting this sysfs parameter to (say) five will shorten the timeout for the *next* stall, or the following warning for the current stall (assuming the stall lasts long enough). It will not affect the timing of the next warning for the current stall.

Stall-warning messages may be enabled and disabled completely via /sys/module/rcupdate/parameters/rcu_cpu_stall_suppress.

CONFIG_RCU_EXP_CPU_STALL_TIMEOUT

Same as the CONFIG_RCU_CPU_STALL_TIMEOUT parameter but only for the expedited grace period. This parameter defines the period of time that RCU will wait from the beginning of an expedited grace period until it issues an RCU CPU stall warning. This time period is normally 20 milliseconds on Android devices. A zero value causes the CONFIG_RCU_CPU_STALL_TIMEOUT value to be used, after conversion to milliseconds.

This configuration parameter may be changed at runtime via the /sys/module/rcupdate/parameters/rcu_exp_cpu_stall_timeout, however this parameter is checked only at the beginning of a cycle. If you are in a current stall cycle, setting it to a new value will change the timeout for the -next- stall.

Stall-warning messages may be enabled and disabled completely via /sys/module/rcupdate/parameters/rcu_cpu_stall_suppress.

RCU_STALL_DELAY_DELTA

Although the lockdep facility is extremely useful, it does add some overhead. Therefore, under CONFIG_PROVE_RCU, the RCU_STALL_DELAY_DELTA macro allows five extra seconds before giving an RCU CPU stall warning message. (This is a cpp macro, not a kernel configuration parameter.)

RCU_STALL_RAT_DELAY

The CPU stall detector tries to make the offending CPU print its own warnings, as this often gives better-quality stack traces. However, if the offending CPU does not detect its own stall in the number of jiffies specified by RCU_STALL_RAT_DELAY, then some other CPU will complain. This delay is normally set to two jiffies. (This is a cpp macro, not a kernel configuration parameter.)

rcupdate.rcu_task_stall_timeout

This boot/sysfs parameter controls the RCU-tasks and RCU-tasks-trace stall warning intervals. A value of zero or less suppresses RCU-tasks stall warnings. A positive value sets the stall-warning interval in seconds. An RCU-tasks stall warning starts with the line:

> INFO: rcu_tasks detected stalls on tasks:

And continues with the output of sched_show_task() for each task stalling the current RCU-tasks grace period.

An RCU-tasks-trace stall warning starts (and continues) similarly:

> INFO: rcu_tasks_trace detected stalls on tasks

Interpreting RCU's CPU Stall-Detector "Splats"

For non-RCU-tasks flavors of RCU, when a CPU detects that some other CPU is stalling, it will print a message similar to the following:

```
INFO: rcu_sched detected stalls on CPUs/tasks:
2-...: (3 GPs behind) idle=06c/0/0 softirq=1453/1455 fqs=0
16-...: (0 ticks this GP) idle=81c/0/0 softirq=764/764 fqs=0
(detected by 32, t=2603 jiffies, g=7075, q=625)
```

This message indicates that CPU 32 detected that CPUs 2 and 16 were both causing stalls, and that the stall was affecting RCU-sched. This message will normally be followed by stack dumps for each CPU. Please note that PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421". It is even possible for an rcu_state stall to be caused by both CPUs *and* tasks, in which case the offending CPUs and tasks will all be called out in the list. In some cases, CPUs will detect themselves stalling, which will result in a self-detected stall.

CPU 2's "(3 GPs behind)" indicates that this CPU has not interacted with the RCU core for the past three grace periods. In contrast, CPU 16's "(0 ticks this GP)" indicates that this CPU has not taken any scheduling-clock interrupts during the current stalled grace period.

The "idle=" portion of the message prints the dyntick-idle state. The hex number before the first "/" is the low-order 12 bits of the dynticks counter, which will have an even-numbered value if the CPU is in dyntick-idle mode and an odd-numbered value otherwise. The hex number between the two "/"s is the value of the nesting, which will be a small non-negative number if in the idle loop (as shown above) and a very large positive number otherwise. The number following the final "/" is the NMI nesting, which will be a small non-negative number.

The "softirq=" portion of the message tracks the number of RCU softirq handlers that the stalled CPU has executed. The number before the "/" is the number that had executed since boot at the time that this CPU last noted the beginning of a grace period, which might be the current (stalled) grace period, or it might be some earlier grace period (for example, if the CPU might have been in dyntick-idle mode for an extended time period). The number after the "/" is the number that have executed since boot until the current time. If this latter number stays constant across repeated stall-warning messages, it is possible that RCU's softirq handlers are no longer able to execute on this CPU. This can happen if the stalled CPU is spinning with interrupts are disabled, or, in -rt kernels, if a high-priority process is starving RCU's softirq handler.

The "fqs=" shows the number of force-quiescent-state idle/offline detection passes that the grace-period kthread has made across this CPU since the last time that this CPU noted the beginning of a grace period.

The "detected by" line indicates which CPU detected the stall (in this case, CPU 32), how many jiffies have elapsed since the start of the grace period (in this case 2603), the grace-period sequence number (7075), and an estimate of the total number of RCU callbacks queued across all CPUs (625 in this case).

If the grace period ends just as the stall warning starts printing, there will be a spurious stall-warning message, which will include the following:

```
INFO: Stall ended before state dump start
```

This is rare, but does happen from time to time in real life. It is also possible for a zero-jiffy stall to be flagged in this case, depending on how the stall warning and the grace-period initialization happen to interact. Please note that it is not possible to entirely eliminate this sort of false positive without resorting to things like stop_machine(), which is overkill for this sort of problem.

If all CPUs and tasks have passed through quiescent states, but the grace period has nevertheless failed to end, the stall-warning splat will include something like the following:

```
All QSes seen, last rcu_preempt kthread activity 23807 (4297905177-4297881370),
↪ jiffies_till_next_fqs=3, root ->qsmask 0x0
```

The "23807" indicates that it has been more than 23 thousand jiffies since the grace-period kthread ran. The "jiffies_till_next_fqs" indicates how frequently that kthread should run, giving the number of jiffies between force-quiescent-state scans, in this case three, which is way less than 23807. Finally, the root rcu_node structure's ->qsmask field is printed, which will normally be zero.

If the relevant grace-period kthread has been unable to run prior to the stall warning, as was the case in the "All QSes seen" line above, the following additional line is printed:

```
rcu_sched kthread starved for 23807 jiffies! g7075 f0x0 RCU_GP_WAIT_FQS(3) ->
↪state=0x1 ->cpu=5
Unless rcu_sched kthread gets sufficient CPU time, OOM is now expected␣
↪behavior.
```

Starving the grace-period kthreads of CPU time can of course result in RCU CPU stall warnings even when all CPUs and tasks have passed through the required quiescent states. The "g" number shows the current grace-period sequence number, the "f" precedes the ->gp_flags command to the grace-period kthread, the "RCU_GP_WAIT_FQS" indicates that the kthread is waiting for a short timeout, the "state" precedes value of the task_struct ->state field, and the "cpu" indicates that the grace-period kthread last ran on CPU 5.

If the relevant grace-period kthread does not wake from FQS wait in a reasonable time, then the following additional line is printed:

```
kthread timer wakeup didn't happen for 23804 jiffies! g7076 f0x0 RCU_GP_WAIT_
↪FQS(5) ->state=0x402
```

The "23804" indicates that kthread's timer expired more than 23 thousand jiffies ago. The rest of the line has meaning similar to the kthread starvation case.

Additionally, the following line is printed:

```
Possible timer handling issue on cpu=4 timer-softirq=11142
```

Here "cpu" indicates that the grace-period kthread last ran on CPU 4, where it queued the fqs timer. The number following the "timer-softirq" is the current `TIMER_SOFTIRQ` count on cpu 4. If this value does not change on successive RCU CPU stall warnings, there is further reason to suspect a timer problem.

These messages are usually followed by stack dumps of the CPUs and tasks involved in the stall. These stack traces can help you locate the cause of the stall, keeping in mind that the CPU detecting the stall will have an interrupt frame that is mainly devoted to detecting the stall.

Multiple Warnings From One Stall

If a stall lasts long enough, multiple stall-warning messages will be printed for it. The second and subsequent messages are printed at longer intervals, so that the time between (say) the first and second message will be about three times the interval between the beginning of the stall and the first message. It can be helpful to compare the stack dumps for the different messages for the same stalled grace period.

Stall Warnings for Expedited Grace Periods

If an expedited grace period detects a stall, it will place a message like the following in dmesg:

```
INFO: rcu_sched detected expedited stalls on CPUs/tasks: { 7-... } 21119␣
↪jiffies s: 73 root: 0x2/.
```

This indicates that CPU 7 has failed to respond to a reschedule IPI. The three periods (".") following the CPU number indicate that the CPU is online (otherwise the first period would instead have been "O"), that the CPU was online at the beginning of the expedited grace period (otherwise the second period would have instead been "o"), and that the CPU has been online at least once since boot (otherwise, the third period would instead have been "N"). The number before the "jiffies" indicates that the expedited grace period has been going on for 21,119 jiffies. The number following the "s:" indicates that the expedited grace-period sequence counter is 73. The fact that this last value is odd indicates that an expedited grace period is in flight. The number following "root:" is a bitmask that indicates which children of the root rcu_node structure correspond to CPUs and/or tasks that are blocking the current expedited grace period. If the tree had more than one level, additional hex numbers would be printed for the states of the other rcu_node structures in the tree.

As with normal grace periods, PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421".

It is entirely possible to see stall warnings from normal and from expedited grace periods at about the same time during the same run.

RCU_CPU_STALL_CPUTIME

In kernels built with CONFIG_RCU_CPU_STALL_CPUTIME=y or booted with rcupdate.rcu_cpu_stall_cputime=1, the following additional information is supplied with each RCU CPU stall warning:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624         45            0
rcu: cputime:       69          1         2425   ==> 2500(ms)
```

These statistics are collected during the sampling period. The values in row "number:" are the number of hard interrupts, number of soft interrupts, and number of context switches on the stalled CPU. The first three values in row "cputime:" indicate the CPU time in milliseconds consumed by hard interrupts, soft interrupts, and tasks on the stalled CPU. The last number is the measurement interval, again in milliseconds. Because user-mode tasks normally do not cause RCU CPU stalls, these tasks are typically kernel tasks, which is why only the system CPU time are considered.

The sampling period is shown as follows:

```
|<------------first timeout---------->|<-----second timeout----->|
|<--half timeout-->|<--half timeout-->|                          |
|                  |<--first period-->|                          |
|                  |<-----------second sampling period---------->|
|                  |                  |                          |
           snapshot time point    1st-stall                  2nd-stall
```

The following describes four typical scenarios:

1. A CPU looping with interrupts disabled.

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:        0          0            0
rcu: cputime:        0          0            0   ==> 2500(ms)
```

   Because interrupts have been disabled throughout the measurement interval, there are no interrupts and no context switches. Furthermore, because CPU time consumption was measured using interrupt handlers, the system CPU consumption is misleadingly measured as zero. This scenario will normally also have "(0 ticks this GP)" printed on this CPU's summary line.

2. A CPU looping with bottom halves disabled.

   This is similar to the previous example, but with non-zero number of and CPU time consumed by hard interrupts, along with non-zero CPU time consumed by in-kernel execution:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624          0            0
rcu: cputime:       49          0         2446   ==> 2500(ms)
```

   The fact that there are zero softirqs gives a hint that these were disabled, perhaps via local_bh_disable(). It is of course possible that there were no softirqs, perhaps because all events that would result in softirq execution are confined to other CPUs. In this case, the diagnosis should continue as shown in the next example.

3. A CPU looping with preemption disabled.

   Here, only the number of context switches is zero:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624         45            0
rcu: cputime:       69          1         2425   ==> 2500(ms)
```

   This situation hints that the stalled CPU was looping with preemption disabled.

4. No looping, but massive hard and soft interrupts.

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:       xx         xx            0
rcu: cputime:       xx         xx            0   ==> 2500(ms)
```

   Here, the number and CPU time of hard interrupts are all non-zero, but the number of context switches and the in-kernel CPU time consumed are zero. The number and cputime of soft interrupts will usually be non-zero, but could be zero, for example, if the CPU was spinning within a single hard interrupt handler.

If this type of RCU CPU stall warning can be reproduced, you can narrow it down by looking at /proc/interrupts or by writing code to trace each interrupt, for example, by referring to show_interrupts().

4.5.12 Using RCU to Protect Read-Mostly Linked Lists

One of the most common uses of RCU is protecting read-mostly linked lists (`struct list_head` in list.h). One big advantage of this approach is that all of the required memory ordering is provided by the list macros. This document describes several list-based RCU use cases.

Example 1: Read-mostly list: Deferred Destruction

A widely used usecase for RCU lists in the kernel is lockless iteration over all processes in the system. `task_struct::tasks` represents the list node that links all the processes. The list can be traversed in parallel to any list additions or removals.

The traversal of the list is done using `for_each_process()` which is defined by the 2 macros:

```
#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)

#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

The code traversing the list of all processes typically looks like:

```
rcu_read_lock();
for_each_process(p) {
        /* Do something with p */
}
rcu_read_unlock();
```

The simplified and heavily inlined code for removing a process from a task list is:

```
void release_task(struct task_struct *p)
{
        write_lock(&tasklist_lock);
        list_del_rcu(&p->tasks);
        write_unlock(&tasklist_lock);
        call_rcu(&p->rcu, delayed_put_task_struct);
}
```

When a process exits, `release_task()` calls `list_del_rcu(&p->tasks)` via __exit_signal() and __unhash_process() under `tasklist_lock` writer lock protection. The list_del_rcu() invocation removes the task from the list of all tasks. The `tasklist_lock` prevents concurrent list additions/removals from corrupting the list. Readers using `for_each_process()` are not protected with the `tasklist_lock`. To prevent readers from noticing changes in the list pointers, the `task_struct` object is freed only after one or more grace periods elapse, with the help of call_rcu(), which is invoked via put_task_struct_rcu_user(). This deferring of destruction ensures that any readers traversing the list will see valid `p->tasks.next` pointers and deletion/freeing can happen in parallel with traversal of the list. This pattern is also called an **existence lock**, since RCU refrains from invoking the delayed_put_task_struct() callback function until all existing readers finish, which guarantees that the `task_struct` object in question will remain in existence until after the completion of all RCU readers that might possibly have a reference to that object.

Example 2: Read-Side Action Taken Outside of Lock: No In-Place Updates

Some reader-writer locking use cases compute a value while holding the read-side lock, but continue to use that value after that lock is released. These use cases are often good candidates for conversion to RCU. One prominent example involves network packet routing. Because the packet-routing data tracks the state of equipment outside of the computer, it will at times contain stale data. Therefore, once the route has been computed, there is no need to hold the routing table static during transmission of the packet. After all, you can hold the routing table static all you want, but that won't keep the external Internet from changing, and it is the state of the external Internet that really matters. In addition, routing entries are typically added or deleted, rather than being modified in place. This is a rare example of the finite speed of light and the non-zero size of atoms actually helping make synchronization be lighter weight.

A straightforward example of this type of RCU use case may be found in the system-call auditing support. For example, a reader-writer locked implementation of `audit_filter_task()` might be as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk, char **key)
{
        struct audit_entry *e;
        enum audit_state   state;

        read_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        read_unlock(&auditsc_lock);
                        return state;
                }
        }
        read_unlock(&auditsc_lock);
        return AUDIT_BUILD_CONTEXT;
}
```

Here the list is searched under the lock, but the lock is dropped before the corresponding value is returned. By the time that this value is acted on, the list may well have been modified. This makes sense, since if you are turning auditing off, it is OK to audit a few extra system calls.

This means that RCU can be easily applied to the read side, as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk, char **key)
{
        struct audit_entry *e;
```

```
        enum audit_state   state;

        rcu_read_lock();
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry_rcu(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        rcu_read_unlock();
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

The read_lock() and read_unlock() calls have become rcu_read_lock() and rcu_read_unlock(), respectively, and the list_for_each_entry() has become list_for_each_entry_rcu(). The **_rcu()** list-traversal primitives add READ_ONCE() and diagnostic checks for incorrect use outside of an RCU read-side critical section.

The changes to the update side are also straightforward. A reader-writer lock might be used as follows for deletion and insertion in these simplified versions of audit_del_rule() and audit_add_rule():

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        write_lock(&auditsc_lock);
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del(&e->list);
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
{
        write_lock(&auditsc_lock);
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add(&entry->list, list);
        } else {
                list_add_tail(&entry->list, list);
```

```
        }
        write_unlock(&auditsc_lock);
        return 0;
}
```

Following are the RCU equivalents for these two functions:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this is the only
         * deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del_rcu(&e->list);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
{
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add_rcu(&entry->list, list);
        } else {
                list_add_tail_rcu(&entry->list, list);
        }
        return 0;
}
```

Normally, the write_lock() and write_unlock() would be replaced by a spin_lock() and a spin_unlock(). But in this case, all callers hold `audit_filter_mutex`, so no additional locking is required. The auditsc_lock can therefore be eliminated, since use of RCU eliminates the need for writers to exclude readers.

The *`list_del()`*, list_add(), and list_add_tail() primitives have been replaced by list_del_rcu(), list_add_rcu(), and *`list_add_tail_rcu()`*. The **_rcu()** list-manipulation primitives add memory barriers that are needed on weakly ordered CPUs. The list_del_rcu() primitive omits the pointer poisoning debug-assist code that would otherwise cause concurrent readers to fail spectacularly.

So, when readers can tolerate stale data and when entries are either added or deleted, without in-place modification, it is very easy to use RCU!

Example 3: Handling In-Place Updates

The system-call auditing code does not update auditing rules in place. However, if it did, the reader-writer-locked code to do so might look as follows (assuming only `field_count` is updated, otherwise, the added fields would need to be filled in):

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        write_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        e->rule.action = newaction;
                        e->rule.field_count = newfield_count;
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}
```

The RCU version creates a copy, updates the copy, then replaces the old entry with the newly updated entry. This sequence of actions, allowing concurrent reads while making a copy to perform an update, is what gives RCU (*read-copy update*) its name.

The RCU version of audit_upd_rule() is as follows:

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        ne = kmalloc(sizeof(*entry), GFP_ATOMIC);
                        if (ne == NULL)
                                return -ENOMEM;
                        audit_copy_rule(&ne->rule, &e->rule);
                        ne->rule.action = newaction;
                        ne->rule.field_count = newfield_count;
                        list_replace_rcu(&e->list, &ne->list);
                        call_rcu(&e->rcu, audit_free_rule);
```

```
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

Again, this assumes that the caller holds `audit_filter_mutex`. Normally, the writer lock would become a spinlock in this sort of code.

The update_lsm_rule() does something very similar, for those who would prefer to look at real Linux-kernel code.

Another use of this pattern can be found in the openswitch driver's *connection tracking table* code in `ct_limit_set()`. The table holds connection tracking entries and has a limit on the maximum entries. There is one such table per-zone and hence one *limit* per zone. The zones are mapped to their limits through a hashtable using an RCU-managed hlist for the hash chains. When a new limit is set, a new limit object is allocated and `ct_limit_set()` is called to replace the old limit object with the new one using *`list_replace_rcu()`*. The old limit object is then freed after a grace period using kfree_rcu().

Example 4: Eliminating Stale Data

The auditing example above tolerates stale data, as do most algorithms that are tracking external state. After all, given there is a delay from the time the external state changes before Linux becomes aware of the change, and so as noted earlier, a small quantity of additional RCU-induced staleness is generally not a problem.

However, there are many examples where stale data cannot be tolerated. One example in the Linux kernel is the System V IPC (see the shm_lock() function in ipc/shm.c). This code checks a *deleted* flag under a per-entry spinlock, and, if the *deleted* flag is set, pretends that the entry does not exist. For this to be helpful, the search function must return holding the per-entry spinlock, as shm_lock() does in fact do.

Quick Quiz:

For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?

Answer to Quick Quiz

If the system-call audit module were to ever need to reject stale data, one way to accomplish this would be to add a `deleted` flag and a `lock` spinlock to the `audit_entry` structure, and modify audit_filter_task() as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
        enum audit_state   state;

        rcu_read_lock();
        list_for_each_entry_rcu(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        spin_lock(&e->lock);
                        if (e->deleted) {
```

```
                                spin_unlock(&e->lock);
                                rcu_read_unlock();
                                return AUDIT_BUILD_CONTEXT;
                        }
                        rcu_read_unlock();
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

The `audit_del_rule()` function would need to set the `deleted` flag under the spinlock as follows:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this
         * is the only deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        spin_lock(&e->lock);
                        list_del_rcu(&e->list);
                        e->deleted = 1;
                        spin_unlock(&e->lock);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

This too assumes that the caller holds `audit_filter_mutex`.

Note that this example assumes that entries are only added and deleted. Additional mechanism is required to deal correctly with the update-in-place performed by audit_upd_rule(). For one thing, audit_upd_rule() would need to hold the locks of both the old `audit_entry` and its replacement while executing the *list_replace_rcu()*.

Example 5: Skipping Stale Objects

For some use cases, reader performance can be improved by skipping stale objects during read-side list traversal, where stale objects are those that will be removed and destroyed after one or more grace periods. One such example can be found in the timerfd subsystem. When a `CLOCK_REALTIME` clock is reprogrammed (for example due to setting of the system time) then all programmed `timerfds` that depend on this clock get triggered and processes waiting on them are awakened in advance of their scheduled expiry. To facilitate this, all such timers are added to an RCU-managed `cancel_list` when they are setup in `timerfd_setup_cancel()`:

```
static void timerfd_setup_cancel(struct timerfd_ctx *ctx, int flags)
{
        spin_lock(&ctx->cancel_lock);
        if ((ctx->clockid == CLOCK_REALTIME ||
             ctx->clockid == CLOCK_REALTIME_ALARM) &&
            (flags & TFD_TIMER_ABSTIME) && (flags & TFD_TIMER_CANCEL_ON_SET)) {
                if (!ctx->might_cancel) {
                        ctx->might_cancel = true;
                        spin_lock(&cancel_lock);
                        list_add_rcu(&ctx->clist, &cancel_list);
                        spin_unlock(&cancel_lock);
                }
        } else {
                __timerfd_remove_cancel(ctx);
        }
        spin_unlock(&ctx->cancel_lock);
}
```

When a timerfd is freed (fd is closed), then the `might_cancel` flag of the timerfd object is cleared, the object removed from the `cancel_list` and destroyed, as shown in this simplified and inlined version of timerfd_release():

```
int timerfd_release(struct inode *inode, struct file *file)
{
        struct timerfd_ctx *ctx = file->private_data;

        spin_lock(&ctx->cancel_lock);
        if (ctx->might_cancel) {
                ctx->might_cancel = false;
                spin_lock(&cancel_lock);
                list_del_rcu(&ctx->clist);
                spin_unlock(&cancel_lock);
        }
        spin_unlock(&ctx->cancel_lock);

        if (isalarm(ctx))
                alarm_cancel(&ctx->t.alarm);
        else
                hrtimer_cancel(&ctx->t.tmr);
        kfree_rcu(ctx, rcu);
        return 0;
}
```

If the CLOCK_REALTIME clock is set, for example by a time server, the hrtimer framework calls timerfd_clock_was_set() which walks the cancel_list and wakes up processes waiting on the timerfd. While iterating the cancel_list, the might_cancel flag is consulted to skip stale objects:

```
void timerfd_clock_was_set(void)
{
        ktime_t moffs = ktime_mono_to_real(0);
        struct timerfd_ctx *ctx;
        unsigned long flags;

        rcu_read_lock();
        list_for_each_entry_rcu(ctx, &cancel_list, clist) {
                if (!ctx->might_cancel)
                        continue;
                spin_lock_irqsave(&ctx->wqh.lock, flags);
                if (ctx->moffs != moffs) {
                        ctx->moffs = KTIME_MAX;
                        ctx->ticks++;
                        wake_up_locked_poll(&ctx->wqh, EPOLLIN);
                }
                spin_unlock_irqrestore(&ctx->wqh.lock, flags);
        }
        rcu_read_unlock();
}
```

The key point is that because RCU-protected traversal of the cancel_list happens concurrently with object addition and removal, sometimes the traversal can access an object that has been removed from the list. In this example, a flag is used to skip such objects.

Summary

Read-mostly list-based data structures that can tolerate stale data are the most amenable to use of RCU. The simplest case is where entries are either added or deleted from the data structure (or atomically modified in place), but non-atomic in-place modifications can be handled by making a copy, updating the copy, then replacing the original with the copy. If stale data cannot be tolerated, then a *deleted* flag may be used in conjunction with a per-entry spinlock in order to allow the search function to reject newly deleted data.

Answer to Quick Quiz:

For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?

If the search function drops the per-entry lock before returning, then the caller will be processing stale data in any case. If it is really OK to be processing stale data, then you don't need a *deleted* flag. If processing stale data really is a problem, then you need to hold the per-entry lock across all of the code that uses the value that was returned.

Back to Quick Quiz

4.5.13 Using RCU to Protect Dynamic NMI Handlers

Although RCU is usually used to protect read-mostly data structures, it is possible to use RCU to provide dynamic non-maskable interrupt handlers, as well as dynamic irq handlers. This document describes how to do this, drawing loosely from Zwane Mwaikambo's NMI-timer work in an old version of "arch/x86/kernel/traps.c".

The relevant pieces of code are listed below, each followed by a brief explanation:

```
static int dummy_nmi_callback(struct pt_regs *regs, int cpu)
{
        return 0;
}
```

The dummy_nmi_callback() function is a "dummy" NMI handler that does nothing, but returns zero, thus saying that it did nothing, allowing the NMI handler to take the default machine-specific action:

```
static nmi_callback_t nmi_callback = dummy_nmi_callback;
```

This nmi_callback variable is a global function pointer to the current NMI handler:

```
void do_nmi(struct pt_regs * regs, long error_code)
{
        int cpu;

        nmi_enter();

        cpu = smp_processor_id();
        ++nmi_count(cpu);

        if (!rcu_dereference_sched(nmi_callback)(regs, cpu))
                default_do_nmi(regs);

        nmi_exit();
}
```

The do_nmi() function processes each NMI. It first disables preemption in the same way that a hardware irq would, then increments the per-CPU count of NMIs. It then invokes the NMI handler stored in the nmi_callback function pointer. If this handler returns zero, do_nmi() invokes the default_do_nmi() function to handle a machine-specific NMI. Finally, preemption is restored.

In theory, *rcu_dereference_sched()* is not needed, since this code runs only on i386, which in theory does not need *rcu_dereference_sched()* anyway. However, in practice it is a good documentation aid, particularly for anyone attempting to do something similar on Alpha or on systems with aggressive optimizing compilers.

Quick Quiz:

Why might the *rcu_dereference_sched()* be necessary on Alpha, given that the code referenced by the pointer is read-only?

Answer to Quick Quiz

Back to the discussion of NMI and RCU:

```
void set_nmi_callback(nmi_callback_t callback)
{
        rcu_assign_pointer(nmi_callback, callback);
}
```

The set_nmi_callback() function registers an NMI handler. Note that any data that is to be used by the callback must be initialized up -before- the call to set_nmi_callback(). On architectures that do not order writes, the rcu_assign_pointer() ensures that the NMI handler sees the initialized values:

```
void unset_nmi_callback(void)
{
        rcu_assign_pointer(nmi_callback, dummy_nmi_callback);
}
```

This function unregisters an NMI handler, restoring the original dummy_nmi_handler(). However, there may well be an NMI handler currently executing on some other CPU. We therefore cannot free up any data structures used by the old NMI handler until execution of it completes on all other CPUs.

One way to accomplish this is via synchronize_rcu(), perhaps as follows:

```
unset_nmi_callback();
synchronize_rcu();
kfree(my_nmi_data);
```

This works because (as of v4.20) synchronize_rcu() blocks until all CPUs complete any preemption-disabled segments of code that they were executing. Since NMI handlers disable preemption, synchronize_rcu() is guaranteed not to return until all ongoing NMI handlers exit. It is therefore safe to free up the handler's data as soon as synchronize_rcu() returns.

Important note: for this to work, the architecture in question must invoke nmi_enter() and nmi_exit() on NMI entry and exit, respectively.

Answer to Quick Quiz:

Why might the `rcu_dereference_sched()` be necessary on Alpha, given that the code referenced by the pointer is read-only?

The caller to set_nmi_callback() might well have initialized some data that is to be used by the new NMI handler. In this case, the `rcu_dereference_sched()` would be needed, because otherwise a CPU that received an NMI just after the new handler was set might see the pointer to the new NMI handler, but the old pre-initialized version of the handler's data.

This same sad story can happen on other CPUs when using a compiler with aggressive pointer-value speculation optimizations. (But please don't!)

More important, the `rcu_dereference_sched()` makes it clear to someone reading the code that the pointer is being protected by RCU-sched.

4.5.14 RCU on Uniprocessor Systems

A common misconception is that, on UP systems, the call_rcu() primitive may immediately invoke its function. The basis of this misconception is that since there is only one CPU, it should not be necessary to wait for anything else to get done, since there are no other CPUs for anything else to be happening on. Although this approach will *sort of* work a surprising amount of the time, it is a very bad idea in general. This document presents three examples that demonstrate exactly how bad an idea this is.

Example 1: softirq Suicide

Suppose that an RCU-based algorithm scans a linked list containing elements A, B, and C in process context, and can delete elements from this same list in softirq context. Suppose that the process-context scan is referencing element B when it is interrupted by softirq processing, which deletes element B, and then invokes call_rcu() to free element B after a grace period.

Now, if call_rcu() were to directly invoke its arguments, then upon return from softirq, the list scan would find itself referencing a newly freed element B. This situation can greatly decrease the life expectancy of your kernel.

This same problem can occur if call_rcu() is invoked from a hardware interrupt handler.

Example 2: Function-Call Fatality

Of course, one could avert the suicide described in the preceding example by having call_rcu() directly invoke its arguments only if it was called from process context. However, this can fail in a similar manner.

Suppose that an RCU-based algorithm again scans a linked list containing elements A, B, and C in process context, but that it invokes a function on each element as it is scanned. Suppose further that this function deletes element B from the list, then passes it to call_rcu() for deferred freeing. This may be a bit unconventional, but it is perfectly legal RCU usage, since call_rcu() must wait for a grace period to elapse. Therefore, in this case, allowing call_rcu() to immediately invoke its arguments would cause it to fail to make the fundamental guarantee underlying RCU, namely that call_rcu() defers invoking its arguments until all RCU read-side critical sections currently executing have completed.

Quick Quiz #1:
Why is it *not* legal to invoke synchronize_rcu() in this case?

Answers to Quick Quiz

Example 3: Death by Deadlock

Suppose that call_rcu() is invoked while holding a lock, and that the callback function must acquire this same lock. In this case, if call_rcu() were to directly invoke the callback, the result would be self-deadlock *even if* this invocation occurred from a later call_rcu() invocation a full grace period later.

In some cases, it would possible to restructure to code so that the call_rcu() is delayed until after the lock is released. However, there are cases where this can be quite ugly:

1. If a number of items need to be passed to call_rcu() within the same critical section, then the code would need to create a list of them, then traverse the list once the lock was released.
2. In some cases, the lock will be held across some kernel API, so that delaying the call_rcu() until the lock is released requires that the data item be passed up via a common API. It is far better to guarantee that callbacks are invoked with no locks held than to have to modify such APIs to allow arbitrary data items to be passed back up through them.

If call_rcu() directly invokes the callback, painful locking restrictions or API changes would be required.

Quick Quiz #2:
What locking restriction must RCU callbacks respect?

Answers to Quick Quiz

It is important to note that userspace RCU implementations *do* permit call_rcu() to directly invoke callbacks, but only if a full grace period has elapsed since those callbacks were queued. This is the case because some userspace environments are extremely constrained. Nevertheless, people writing userspace RCU implementations are strongly encouraged to avoid invoking callbacks from call_rcu(), thus obtaining the deadlock-avoidance benefits called out above.

Summary

Permitting call_rcu() to immediately invoke its arguments breaks RCU, even on a UP system. So do not do it! Even on a UP system, the RCU infrastructure *must* respect grace periods, and *must* invoke callbacks from a known environment in which no locks are held.

Note that it *is* safe for synchronize_rcu() to return immediately on UP systems, including PREEMPT SMP builds running on UP systems.

Quick Quiz #3:
Why can't synchronize_rcu() return immediately on UP systems running preemptible RCU?

Answer to Quick Quiz #1:
Why is it *not* legal to invoke synchronize_rcu() in this case?

Because the calling function is scanning an RCU-protected linked list, and is therefore within an RCU read-side critical section. Therefore, the called function has been invoked within an RCU read-side critical section, and is not permitted to block.

Answer to Quick Quiz #2:
What locking restriction must RCU callbacks respect?

Any lock that is acquired within an RCU callback must be acquired elsewhere using an _bh variant of the spinlock primitive. For example, if "mylock" is acquired by an RCU callback, then a process-context acquisition of this lock must use something like spin_lock_bh() to acquire the lock. Please note that it is also OK to use _irq variants of spinlocks, for example, spin_lock_irqsave().

If the process-context code were to simply use spin_lock(), then, since RCU callbacks can be invoked from softirq context, the callback might be called from a softirq that interrupted the process-context critical section. This would result in self-deadlock.

This restriction might seem gratuitous, since very few RCU callbacks acquire locks directly. However, a great many RCU callbacks do acquire locks *indirectly*, for example, via the kfree() primitive.

Answer to Quick Quiz #3:

Why can't synchronize_rcu() return immediately on UP systems running preemptible RCU?

Because some other task might have been preempted in the middle of an RCU read-side critical section. If synchronize_rcu() simply immediately returned, it would prematurely signal the end of the grace period, which would come as a nasty shock to that other thread when it started running again.

4.5.15 A Tour Through TREE_RCU's Grace-Period Memory Ordering

August 8, 2017

This article was contributed by Paul E. McKenney

Introduction

This document gives a rough visual overview of how Tree RCU's grace-period memory ordering guarantee is provided.

What Is Tree RCU's Grace Period Memory Ordering Guarantee?

RCU grace periods provide extremely strong memory-ordering guarantees for non-idle non-offline code. Any code that happens after the end of a given RCU grace period is guaranteed to see the effects of all accesses prior to the beginning of that grace period that are within RCU read-side critical sections. Similarly, any code that happens before the beginning of a given RCU grace period is guaranteed to not see the effects of all accesses following the end of that grace period that are within RCU read-side critical sections.

Note well that RCU-sched read-side critical sections include any region of code for which preemption is disabled. Given that each individual machine instruction can be thought of as an extremely small region of preemption-disabled code, one can think of `synchronize_rcu()` as `smp_mb()` on steroids.

RCU updaters use this guarantee by splitting their updates into two phases, one of which is executed before the grace period and the other of which is executed after the grace period. In the most common use case, phase one removes an element from a linked RCU-protected data structure, and phase two frees that element. For this to work, any readers that have witnessed state prior to the phase-one update (in the common case, removal) must not witness state following the phase-two update (in the common case, freeing).

The RCU implementation provides this guarantee using a network of lock-based critical sections, memory barriers, and per-CPU processing, as is described in the following sections.

Tree RCU Grace Period Memory Ordering Building Blocks

The workhorse for RCU's grace-period memory ordering is the critical section for the `rcu_node` structure's `->lock`. These critical sections use helper functions for lock acquisition, including `raw_spin_lock_rcu_node()`, `raw_spin_lock_irq_rcu_node()`, and `raw_spin_lock_irqsave_rcu_node()`. Their lock-release counterparts are `raw_spin_unlock_rcu_node()`, `raw_spin_unlock_irq_rcu_node()`, and `raw_spin_unlock_irqrestore_rcu_node()`, respectively. For completeness, a `raw_spin_trylock_rcu_node()` is also provided. The key point is that the lock-acquisition functions, including `raw_spin_trylock_rcu_node()`, all invoke `smp_mb__after_unlock_lock()` immediately after successful acquisition of the lock.

Therefore, for any given `rcu_node` structure, any access happening before one of the above lock-release functions will be seen by all CPUs as happening before any access happening after a later one of the above lock-acquisition functions. Furthermore, any access happening before one of the above lock-release function on any given CPU will be seen by all CPUs as happening before any access happening after a later one of the above lock-acquisition functions executing on that same CPU, even if the lock-release and lock-acquisition functions are operating on different `rcu_node` structures. Tree RCU uses these two ordering guarantees to form an ordering network among all CPUs that were in any way involved in the grace period, including any CPUs that came online or went offline during the grace period in question.

The following litmus test exhibits the ordering effects of these lock-acquisition and lock-release functions:

```
int x, y, z;

void task0(void)
{
  raw_spin_lock_rcu_node(rnp);
  WRITE_ONCE(x, 1);
  r1 = READ_ONCE(y);
  raw_spin_unlock_rcu_node(rnp);
}

void task1(void)
{
  raw_spin_lock_rcu_node(rnp);
  WRITE_ONCE(y, 1);
  r2 = READ_ONCE(z);
  raw_spin_unlock_rcu_node(rnp);
}

void task2(void)
{
  WRITE_ONCE(z, 1);
  smp_mb();
  r3 = READ_ONCE(x);
}

WARN_ON(r1 == 0 && r2 == 0 && r3 == 0);
```

The `WARN_ON()` is evaluated at "the end of time", after all changes have propagated throughout the system. Without the `smp_mb__after_unlock_lock()` provided by the acquisition functions, this `WARN_ON()` could trigger, for example on PowerPC. The `smp_mb__after_unlock_lock()` invocations prevent this `WARN_ON()` from triggering.

Quick Quiz:

But the chain of rcu_node-structure lock acquisitions guarantees that new readers will see all of the updater's pre-grace-period accesses and also guarantees that the updater's post-grace-period accesses will see all of the old reader's accesses. So why do we need all of those calls to smp_mb__after_unlock_lock()?

Answer:

Because we must provide ordering for RCU's polling grace-period primitives, for example, *get_state_synchronize_rcu()* and *poll_state_synchronize_rcu()*. Consider this code:

```
CPU 0                                     CPU 1
----                                      ----
WRITE_ONCE(X, 1)                          WRITE_ONCE(Y, 1)
g = get_state_synchronize_rcu()           smp_mb()
while (!poll_state_synchronize_rcu(g))    r1 = READ_ONCE(X)
        continue;
r0 = READ_ONCE(Y)
```

RCU guarantees that the outcome r0 == 0 && r1 == 0 will not happen, even if CPU 1 is in an RCU extended quiescent state (idle or offline) and thus won't interact directly with the RCU core processing at all.

This approach must be extended to include idle CPUs, which need RCU's grace-period memory ordering guarantee to extend to any RCU read-side critical sections preceding and following the current idle sojourn. This case is handled by calls to the strongly ordered `atomic_add_return()` read-modify-write atomic operation that is invoked within `rcu_dynticks_eqs_enter()` at idle-entry time and within `rcu_dynticks_eqs_exit()` at idle-exit time. The grace-period kthread invokes `rcu_dynticks_snap()` and `rcu_dynticks_in_eqs_since()` (both of which invoke an `atomic_add_return()` of zero) to detect idle CPUs.

Quick Quiz:

But what about CPUs that remain offline for the entire grace period?

Answer:

Such CPUs will be offline at the beginning of the grace period, so the grace period won't expect quiescent states from them. Races between grace-period start and CPU-hotplug operations are mediated by the CPU's leaf `rcu_node` structure's `->lock` as described above.

The approach must be extended to handle one final case, that of waking a task blocked in `synchronize_rcu()`. This task might be affined to a CPU that is not yet aware that the grace period has ended, and thus might not yet be subject to the grace period's memory ordering. Therefore, there is an `smp_mb()` after the return from `wait_for_completion()` in the `synchronize_rcu()` code path.

Quick Quiz:

What? Where??? I don't see any `smp_mb()` after the return from `wait_for_completion()`!!!

Answer:

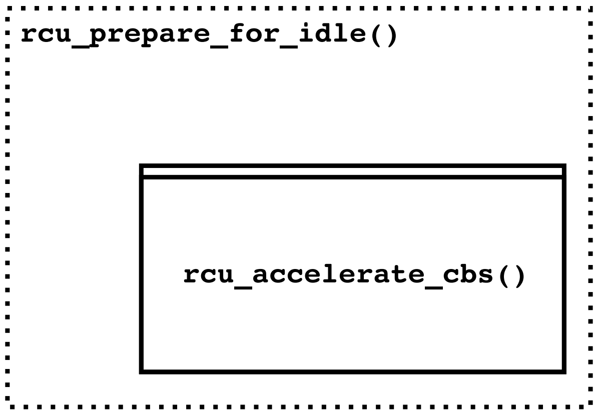
That would be because I spotted the need for that `smp_mb()` during the creation of this documentation, and it is therefore unlikely to hit mainline before v4.14. Kudos to Lance Roy, Will Deacon, Peter Zijlstra, and Jonathan Cameron for asking questions that sensitized me to the rather elaborate sequence of events that demonstrate the need for this memory barrier.

Tree RCU's grace--period memory-ordering guarantees rely most heavily on the `rcu_node` structure's `->lock` field, so much so that it is necessary to abbreviate this pattern in the diagrams in the next section. For example, consider the `rcu_prepare_for_idle()` function shown below, which is one of several functions that enforce ordering of newly arrived RCU callbacks against future grace periods:

```
static void rcu_prepare_for_idle(void)
{
  bool needwake;
  struct rcu_data *rdp = this_cpu_ptr(&rcu_data);
  struct rcu_node *rnp;
  int tne;

  lockdep_assert_irqs_disabled();
  if (rcu_rdp_is_offloaded(rdp))
    return;

  /* Handle nohz enablement switches conservatively. */
  tne = READ_ONCE(tick_nohz_active);
  if (tne != rdp->tick_nohz_enabled_snap) {
    if (!rcu_segcblist_empty(&rdp->cblist))
      invoke_rcu_core(); /* force nohz to see update. */
    rdp->tick_nohz_enabled_snap = tne;
    return;
  }
  if (!tne)
    return;

  /*
   * If we have not yet accelerated this jiffy, accelerate all
   * callbacks on this CPU.
  */
  if (rdp->last_accelerate == jiffies)
    return;
  rdp->last_accelerate = jiffies;
  if (rcu_segcblist_pend_cbs(&rdp->cblist)) {
    rnp = rdp->mynode;
    raw_spin_lock_rcu_node(rnp); /* irqs already disabled. */
    needwake = rcu_accelerate_cbs(rnp, rdp);
    raw_spin_unlock_rcu_node(rnp); /* irqs remain disabled. */
    if (needwake)
      rcu_gp_kthread_wake();
```

```
37   }
38 }
```

But the only part of `rcu_prepare_for_idle()` that really matters for this discussion are lines 32–34. We will therefore abbreviate this function as follows:



The box represents the `rcu_node` structure's `->lock` critical section, with the double line on top representing the additional `smp_mb__after_unlock_lock()`.

Tree RCU Grace Period Memory Ordering Components

Tree RCU's grace-period memory-ordering guarantee is provided by a number of RCU components:

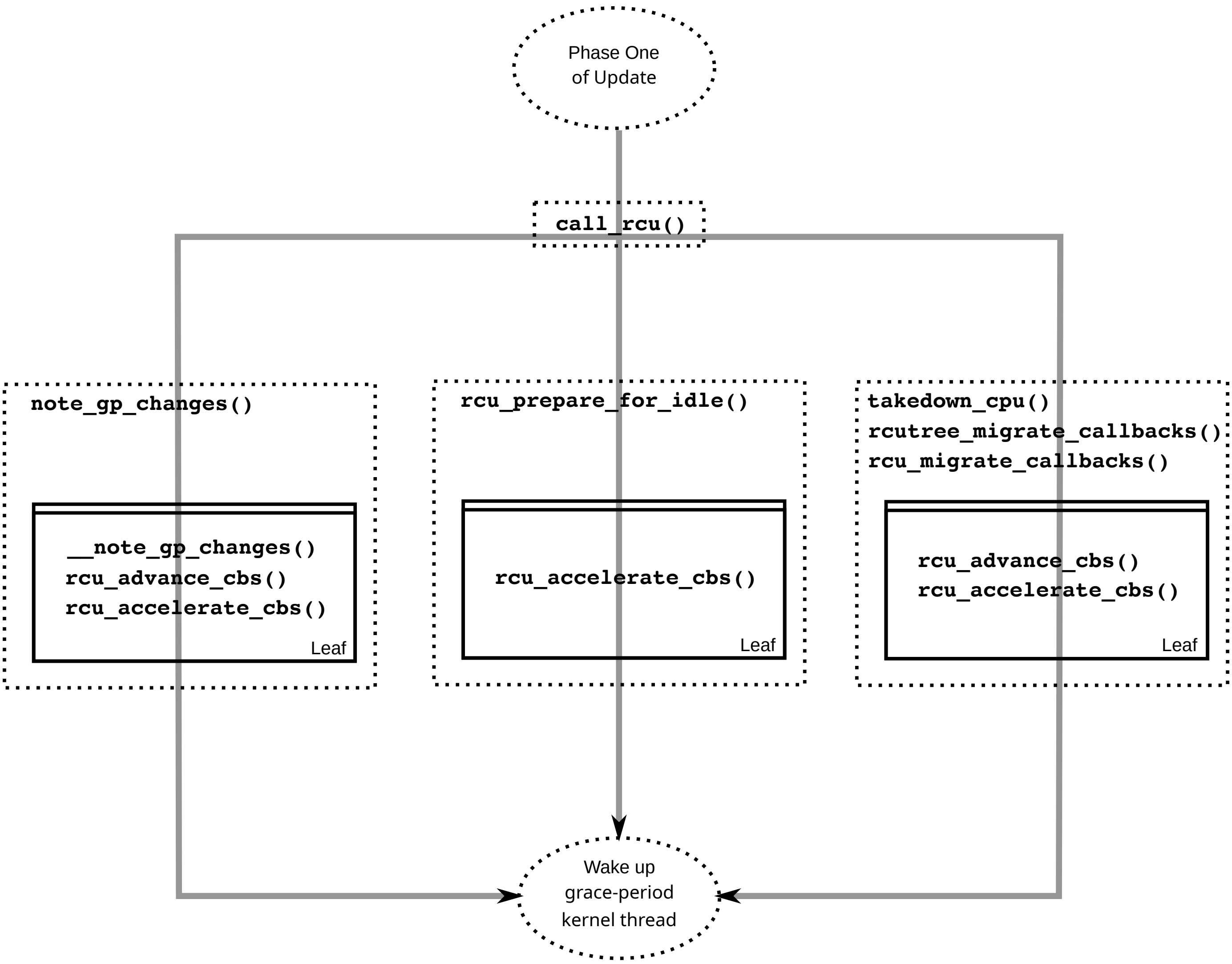
1. *Callback Registry*
2. *Grace-Period Initialization*
3. *Self-Reported Quiescent States*
4. *Dynamic Tick Interface*
5. *CPU-Hotplug Interface*
6. *Forcing Quiescent States*
7. *Grace-Period Cleanup*
8. *Callback Invocation*

Each of the following section looks at the corresponding component in detail.

Callback Registry

If RCU's grace-period guarantee is to mean anything at all, any access that happens before a given invocation of `call_rcu()` must also happen before the corresponding grace period. The implementation of this portion of RCU's grace period guarantee is shown in the following figure:

Because `call_rcu()` normally acts only on CPU-local state, it provides no ordering guarantees, either for itself or for phase one of the update (which again will usually be removal of an element from an RCU-protected data structure). It simply enqueues the `rcu_head` structure


Phase One
of Update
call_rcu()
note_gp_changes()
__note_gp_changes()
rcu_advance_cbs()
rcu_accelerate_cbs()
Leaf
rcu_prepare_for_idle()
rcu_accelerate_cbs()
Leaf
takedown_cpu()
rcutree_migrate_callbacks()
rcu_migrate_callbacks()
rcu_advance_cbs()
rcu_accelerate_cbs()
Leaf
Wake up
grace-period
kernel thread

on a per-CPU list, which cannot become associated with a grace period until a later call to `rcu_accelerate_cbs()`, as shown in the diagram above.

One set of code paths shown on the left invokes `rcu_accelerate_cbs()` via `note_gp_changes()`, either directly from `call_rcu()` (if the current CPU is inundated with queued `rcu_head` structures) or more likely from an `RCU_SOFTIRQ` handler. Another code path in the middle is taken only in kernels built with `CONFIG_RCU_FAST_NO_HZ=y`, which invokes `rcu_accelerate_cbs()` via `rcu_prepare_for_idle()`. The final code path on the right is taken only in kernels built with `CONFIG_HOTPLUG_CPU=y`, which invokes `rcu_accelerate_cbs()` via `rcu_advance_cbs()`, `rcu_migrate_callbacks`, `rcutree_migrate_callbacks()`, and `takedown_cpu()`, which in turn is invoked on a surviving CPU after the outgoing CPU has been completely offlined.

There are a few other code paths within grace-period processing that opportunistically invoke `rcu_accelerate_cbs()`. However, either way, all of the CPU's recently queued `rcu_head` structures are associated with a future grace-period number under the protection of the CPU's lead `rcu_node` structure's `->lock`. In all cases, there is full ordering against any prior critical section for that same `rcu_node` structure's `->lock`, and also full ordering against any of the current task's or CPU's prior critical sections for any `rcu_node` structure's `->lock`.

The next section will show how this ordering ensures that any accesses prior to the `call_rcu()` (particularly including phase one of the update) happen before the start of the corresponding grace period.

Quick Quiz:
But what about `synchronize_rcu()`?
Answer:
The `synchronize_rcu()` passes `call_rcu()` to `wait_rcu_gp()`, which invokes it. So either way, it eventually comes down to `call_rcu()`.

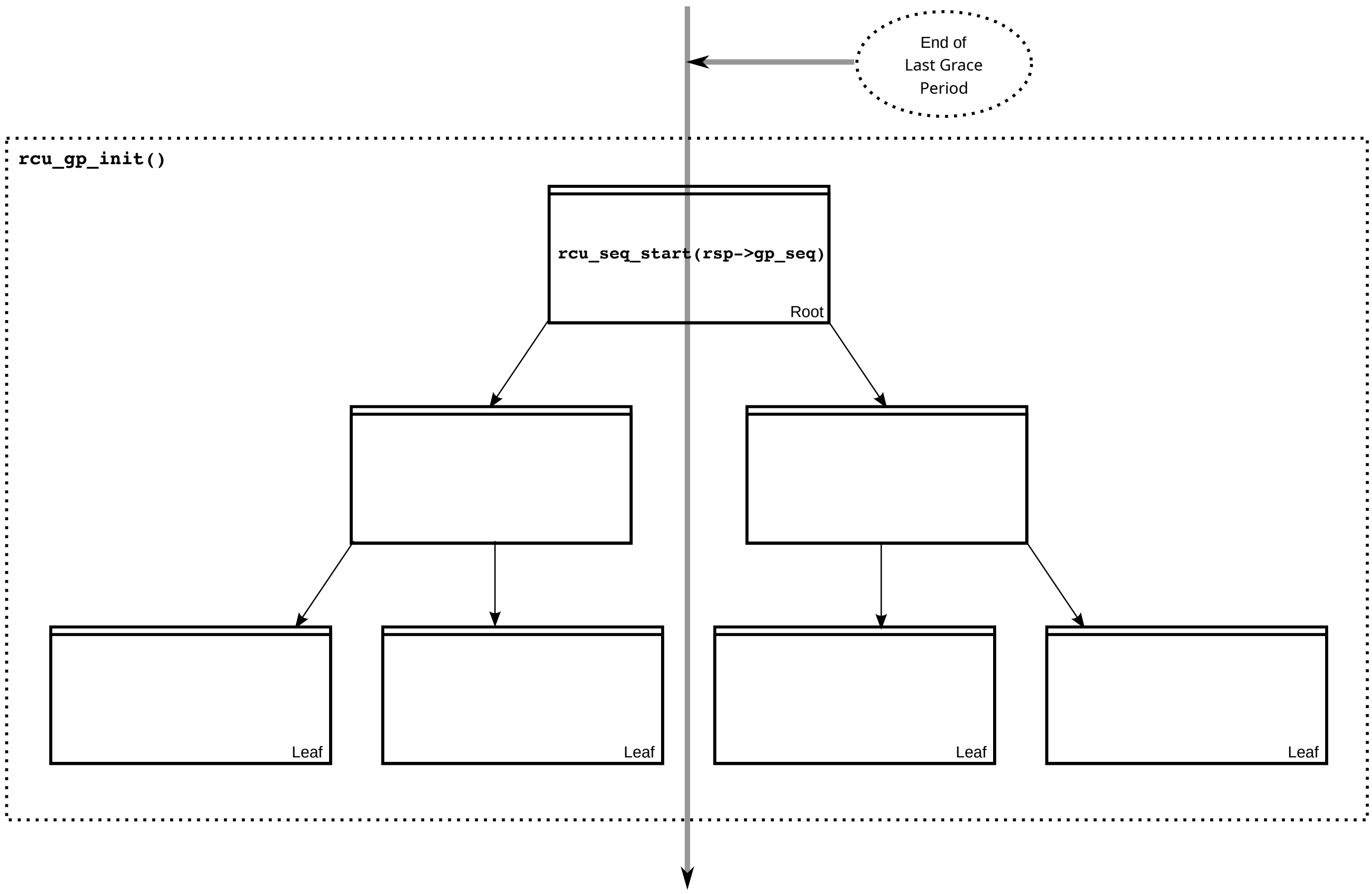
Grace-Period Initialization

Grace-period initialization is carried out by the grace-period kernel thread, which makes several passes over the `rcu_node` tree within the `rcu_gp_init()` function. This means that showing the full flow of ordering through the grace-period computation will require duplicating this tree. If you find this confusing, please note that the state of the `rcu_node` changes over time, just like Heraclitus's river. However, to keep the `rcu_node` river tractable, the grace-period kernel thread's traversals are presented in multiple parts, starting in this section with the various phases of grace-period initialization.

The first ordering-related grace-period initialization action is to advance the `rcu_state` structure's `->gp_seq` grace-period-number counter, as shown below:

The actual increment is carried out using `smp_store_release()`, which helps reject false-positive RCU CPU stall detection. Note that only the root `rcu_node` structure is touched.

The first pass through the `rcu_node` tree updates bitmasks based on CPUs having come online or gone offline since the start of the previous grace period. In the common case where the number of online CPUs for this `rcu_node` structure has not transitioned to or from zero, this pass will scan only the leaf `rcu_node` structures. However, if the number of online CPUs for a given leaf `rcu_node` structure has transitioned from zero, `rcu_init_new_rnp()` will be invoked for the first incoming CPU. Similarly, if the number of online CPUs for a given leaf `rcu_node` structure



has transitioned to zero, `rcu_cleanup_dead_rnp()` will be invoked for the last outgoing CPU. The diagram below shows the path of ordering if the leftmost `rcu_node` structure onlines its first CPU and if the next `rcu_node` structure has no online CPUs (or, alternatively if the leftmost `rcu_node` structure offlines its last CPU and if the next `rcu_node` structure has no online CPUs).

The final `rcu_gp_init()` pass through the `rcu_node` tree traverses breadth-first, setting each `rcu_node` structure's `->gp_seq` field to the newly advanced value from the `rcu_state` structure, as shown in the following diagram.

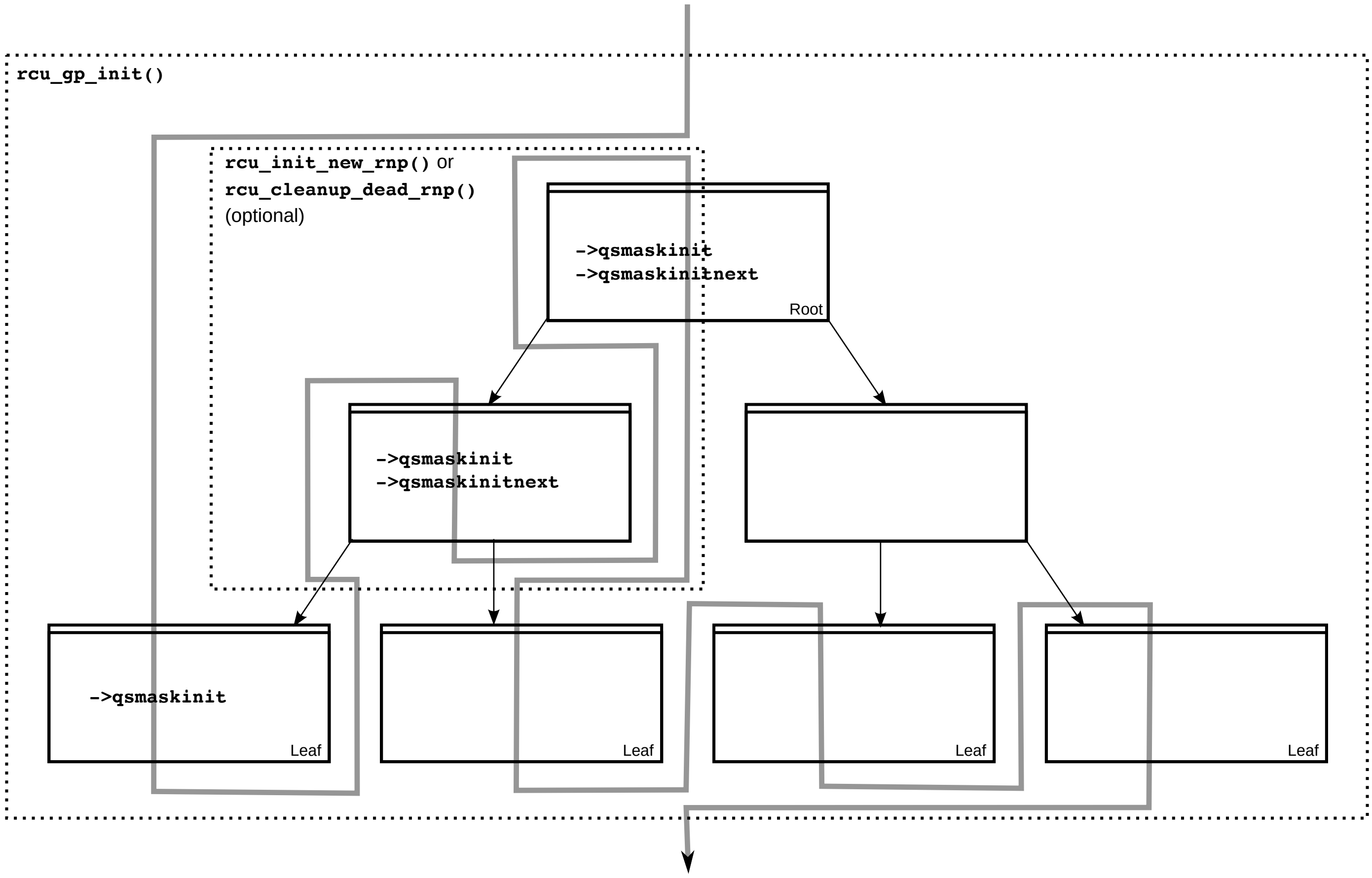
This change will also cause each CPU's next call to `__note_gp_changes()` to notice that a new grace period has started, as described in the next section. But because the grace-period kthread started the grace period at the root (with the advancing of the `rcu_state` structure's `->gp_seq` field) before setting each leaf `rcu_node` structure's `->gp_seq` field, each CPU's observation of the start of the grace period will happen after the actual start of the grace period.

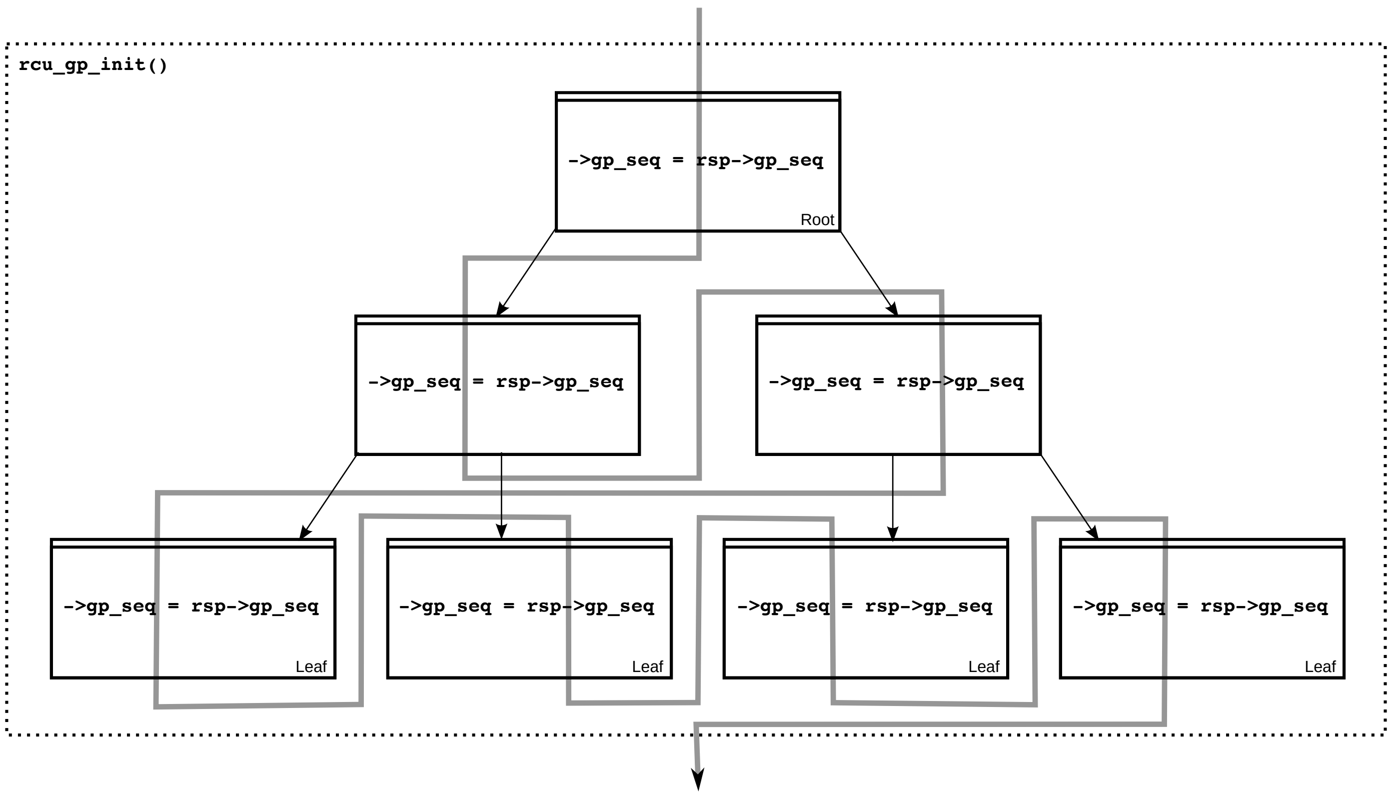
Quick Quiz:

But what about the CPU that started the grace period? Why wouldn't it see the start of the grace period right when it started that grace period?

Answer:

In some deep philosophical and overly anthromorphized sense, yes, the CPU starting the grace period is immediately aware of having done so. However, if we instead assume that RCU is not self-aware, then even the CPU starting the grace period does not really become aware of the start of this grace period until its first call to `__note_gp_changes()`. On the other hand, this CPU potentially gets early notification because it invokes `__note_gp_changes()` during its last `rcu_gp_init()` pass through its leaf `rcu_node` structure.


rcu_gp_init()
rcu_init_new_rnp() or
rcu_cleanup_dead_rnp()
(optional)
->qsmaskinit
->qsmaskinitnext
Root
->qsmaskinit
->qsmaskinitnext
->qsmaskinit
Leaf
Leaf
Leaf
Leaf


rcu_gp_init()
->gp_seq = rsp->gp_seq
Root
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
Leaf
Leaf
Leaf
Leaf

Self-Reported Quiescent States

When all entities that might block the grace period have reported quiescent states (or as described in a later section, had quiescent states reported on their behalf), the grace period can end. Online non-idle CPUs report their own quiescent states, as shown in the following diagram:

This is for the last CPU to report a quiescent state, which signals the end of the grace period. Earlier quiescent states would push up the `rcu_node` tree only until they encountered an `rcu_node` structure that is waiting for additional quiescent states. However, ordering is nevertheless preserved because some later quiescent state will acquire that `rcu_node` structure's `->lock`.

Any number of events can lead up to a CPU invoking `note_gp_changes` (or alternatively, directly invoking `__note_gp_changes()`), at which point that CPU will notice the start of a new grace period while holding its leaf `rcu_node` lock. Therefore, all execution shown in this diagram happens after the start of the grace period. In addition, this CPU will consider any RCU read-side critical section that started before the invocation of `__note_gp_changes()` to have started before the grace period, and thus a critical section that the grace period must wait on.

Quick Quiz:

But a RCU read-side critical section might have started after the beginning of the grace period (the advancing of `->gp_seq` from earlier), so why should the grace period wait on such a critical section?

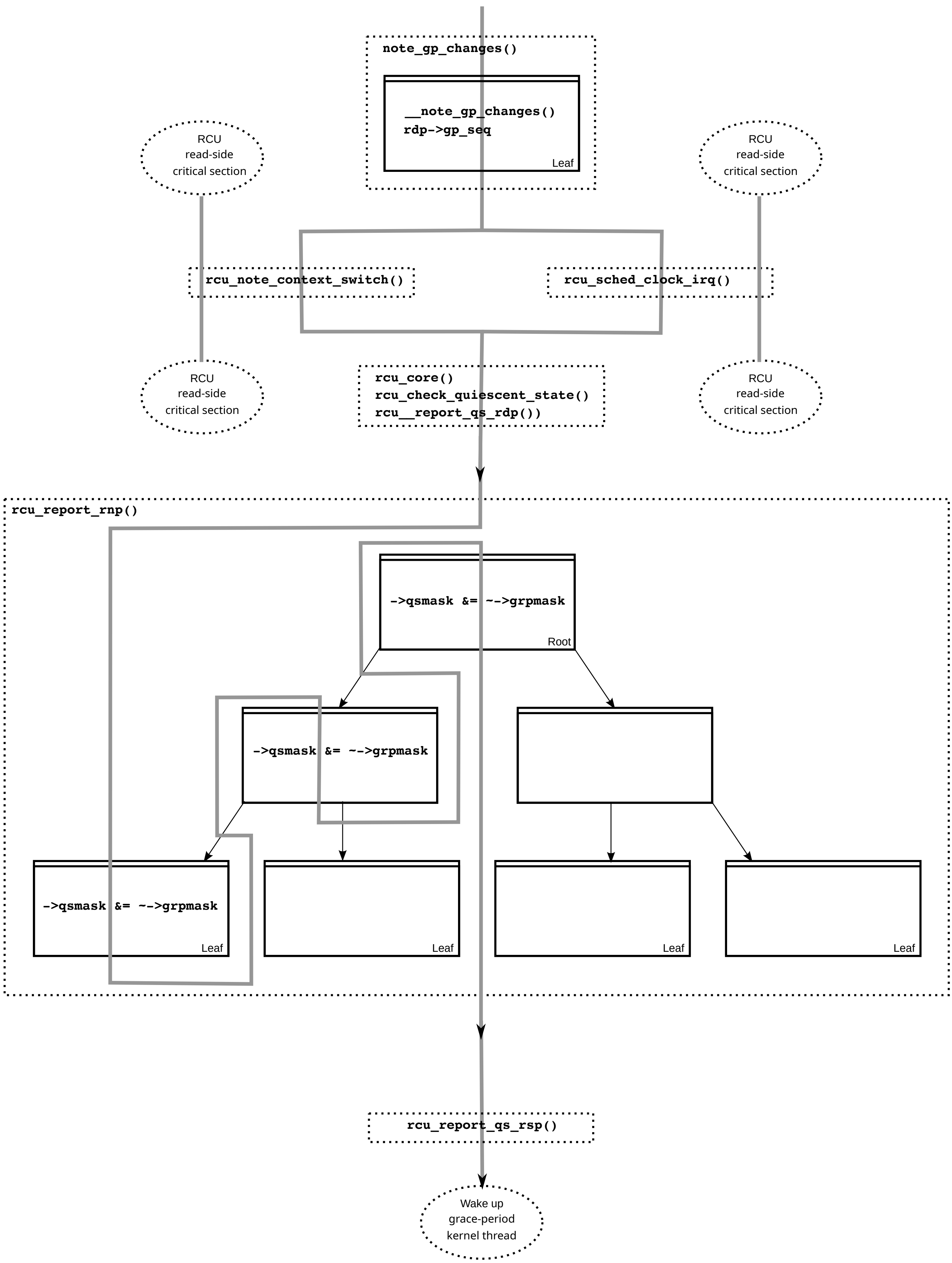
Answer:

It is indeed not necessary for the grace period to wait on such a critical section. However, it is permissible to wait on it. And it is furthermore important to wait on it, as this lazy approach is far more scalable than a "big bang" all-at-once grace-period start could possibly be.

If the CPU does a context switch, a quiescent state will be noted by `rcu_note_context_switch()` on the left. On the other hand, if the CPU takes a scheduler-clock interrupt while executing in usermode, a quiescent state will be noted by `rcu_sched_clock_irq()` on the right. Either way, the passage through a quiescent state will be noted in a per-CPU variable.

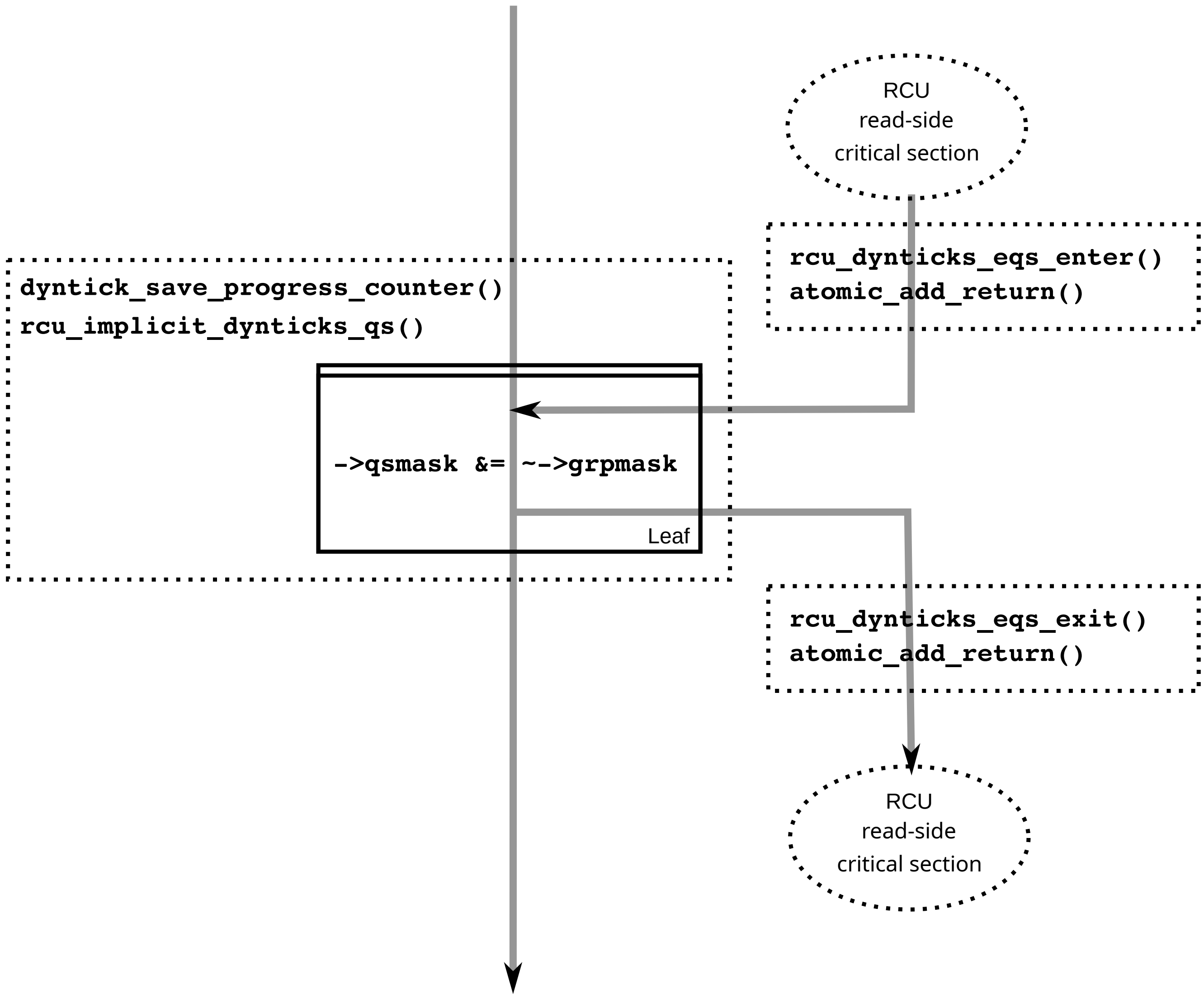
The next time an `RCU_SOFTIRQ` handler executes on this CPU (for example, after the next scheduler-clock interrupt), `rcu_core()` will invoke `rcu_check_quiescent_state()`, which will notice the recorded quiescent state, and invoke `rcu_report_qs_rdp()`. If `rcu_report_qs_rdp()` verifies that the quiescent state really does apply to the current grace period, it invokes `rcu_report_rnp()` which traverses up the `rcu_node` tree as shown at the bottom of the diagram, clearing bits from each `rcu_node` structure's `->qsmask` field, and propagating up the tree when the result is zero.

Note that traversal passes upwards out of a given `rcu_node` structure only if the current CPU is reporting the last quiescent state for the subtree headed by that `rcu_node` structure. A key point is that if a CPU's traversal stops at a given `rcu_node` structure, then there will be a later traversal by another CPU (or perhaps the same one) that proceeds upwards from that point, and the `rcu_node` `->lock` guarantees that the first CPU's quiescent state happens before the remainder of the second CPU's traversal. Applying this line of thought repeatedly shows that all CPUs' quiescent states happen before the last CPU traverses through the root `rcu_node` structure, the "last CPU" being the one that clears the last bit in the root `rcu_node` structure's `->qsmask` field.


note_gp_changes()
__note_gp_changes()
rdp->gp_seq
Leaf
RCU
read-side
critical section
RCU
read-side
critical section
rcu_note_context_switch()
rcu_sched_clock_irq()
RCU
read-side
critical section
rcu_core()
rcu_check_quiescent_state()
rcu__report_qs_rdp())
RCU
read-side
critical section
rcu_report_rnp()
->qsmask &= ~->grpmask
Root
->qsmask &= ~->grpmask
->qsmask &= ~->grpmask
Leaf
Leaf
Leaf
Leaf
rcu_report_qs_rsp()
Wake up
grace-period
kernel thread

Dynamic Tick Interface

Due to energy-efficiency considerations, RCU is forbidden from disturbing idle CPUs. CPUs are therefore required to notify RCU when entering or leaving idle state, which they do via fully ordered value-returning atomic operations on a per-CPU variable. The ordering effects are as shown below:

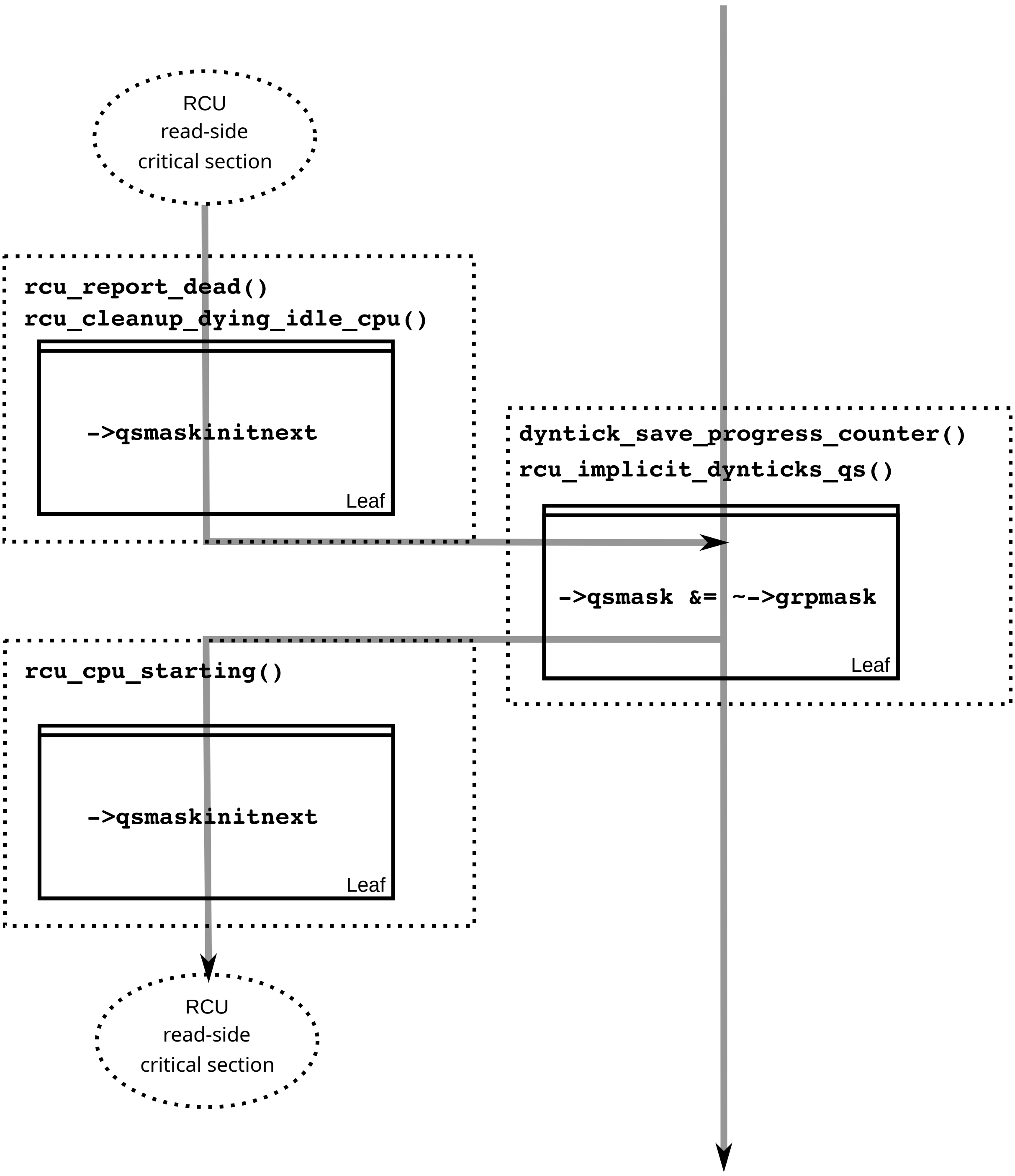


The RCU grace-period kernel thread samples the per-CPU idleness variable while holding the corresponding CPU's leaf `rcu_node` structure's `->lock`. This means that any RCU read-side critical sections that precede the idle period (the oval near the top of the diagram above) will happen before the end of the current grace period. Similarly, the beginning of the current grace period will happen before any RCU read-side critical sections that follow the idle period (the oval near the bottom of the diagram above).

Plumbing this into the full grace-period execution is described *below*.

CPU-Hotplug Interface

RCU is also forbidden from disturbing offline CPUs, which might well be powered off and removed from the system completely. CPUs are therefore required to notify RCU of their comings and goings as part of the corresponding CPU hotplug operations. The ordering effects are shown below:

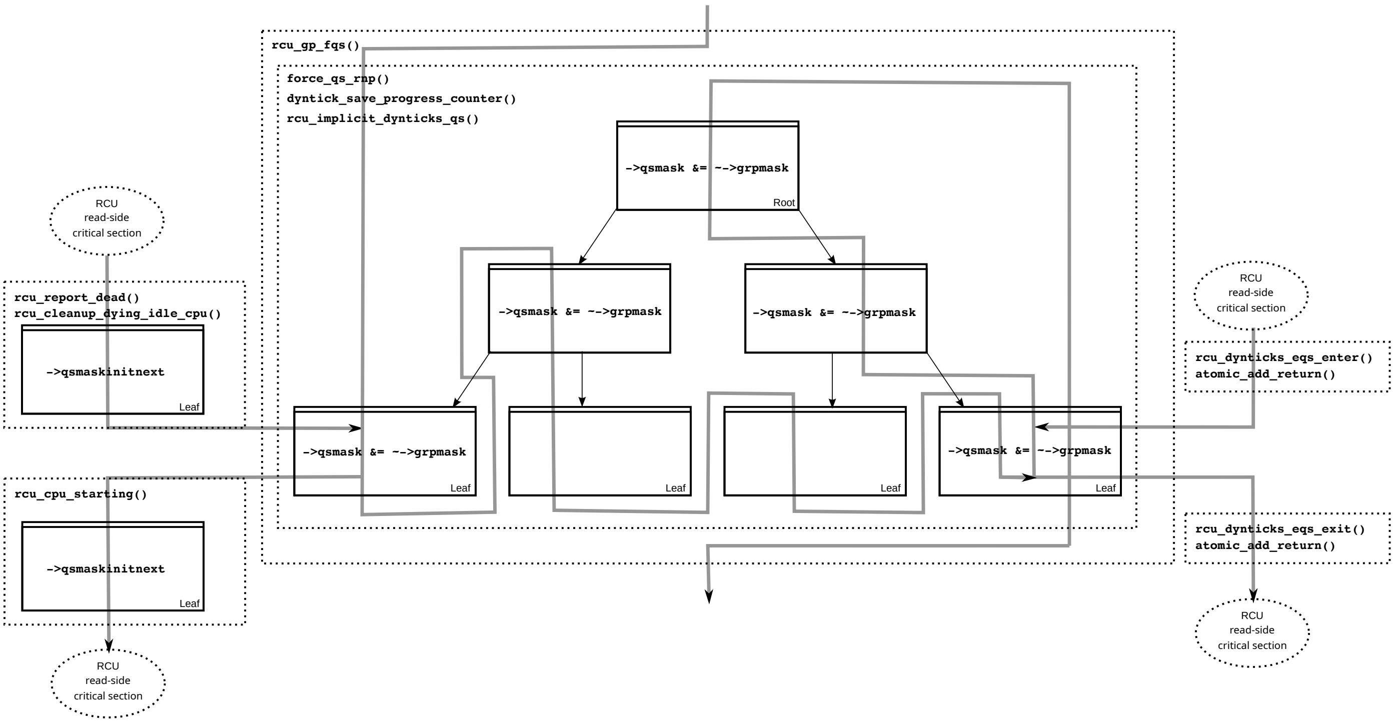


Because CPU hotplug operations are much less frequent than idle transitions, they are heavier weight, and thus acquire the CPU's leaf `rcu_node` structure's `->lock` and update this structure's `->qsmaskinitnext`. The RCU grace-period kernel thread samples this mask to detect CPUs having gone offline since the beginning of this grace period.

Plumbing this into the full grace-period execution is described *below*.

Forcing Quiescent States

As noted above, idle and offline CPUs cannot report their own quiescent states, and therefore the grace-period kernel thread must do the reporting on their behalf. This process is called "forcing quiescent states", it is repeated every few jiffies, and its ordering effects are shown below:



Each pass of quiescent state forcing is guaranteed to traverse the leaf `rcu_node` structures, and if there are no new quiescent states due to recently idled and/or offlined CPUs, then only the leaves are traversed. However, if there is a newly offlined CPU as illustrated on the left or a newly idled CPU as illustrated on the right, the corresponding quiescent state will be driven up towards the root. As with self-reported quiescent states, the upwards driving stops once it reaches an `rcu_node` structure that has quiescent states outstanding from other CPUs.

Quick Quiz:

The leftmost drive to root stopped before it reached the root `rcu_node` structure, which means that there are still CPUs subordinate to that structure on which the current grace period is waiting. Given that, how is it possible that the rightmost drive to root ended the grace period?

Answer:

Good analysis! It is in fact impossible in the absence of bugs in RCU. But this diagram is complex enough as it is, so simplicity overrode accuracy. You can think of it as poetic license, or you can think of it as misdirection that is resolved in the *stitched-together diagram*.

Grace-Period Cleanup

Grace-period cleanup first scans the rcu_node tree breadth-first advancing all the ->gp_seq fields, then it advances the rcu_state structure's ->gp_seq field. The ordering effects are shown below:

As indicated by the oval at the bottom of the diagram, once grace-period cleanup is complete, the next grace period can begin.

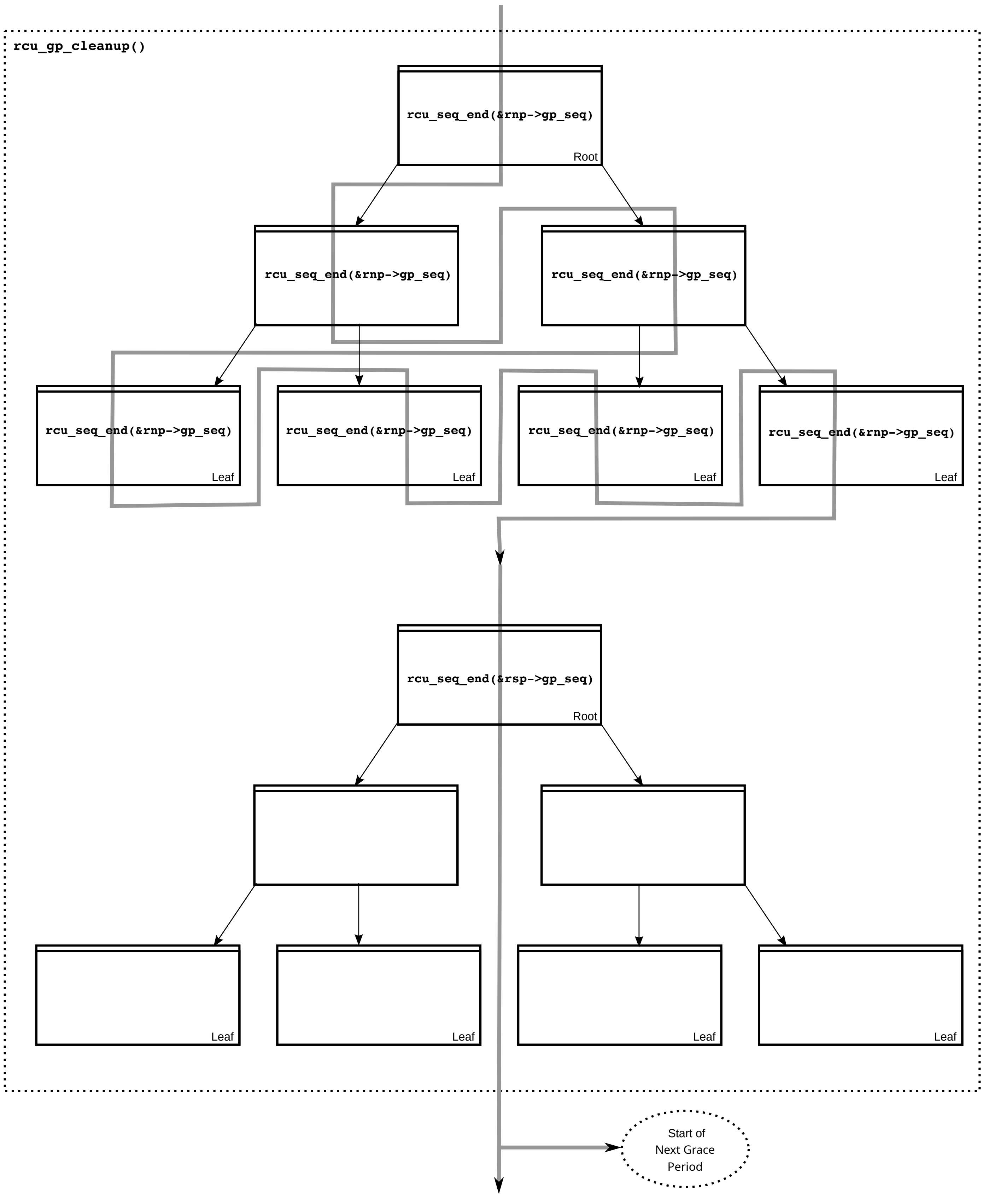
Quick Quiz:
But when precisely does the grace period end?
Answer:
There is no useful single point at which the grace period can be said to end. The earliest reasonable candidate is as soon as the last CPU has reported its quiescent state, but it may be some milliseconds before RCU becomes aware of this. The latest reasonable candidate is once the rcu_state structure's ->gp_seq field has been updated, but it is quite possible that some CPUs have already completed phase two of their updates by that time. In short, if you are going to work with RCU, you need to learn to embrace uncertainty.

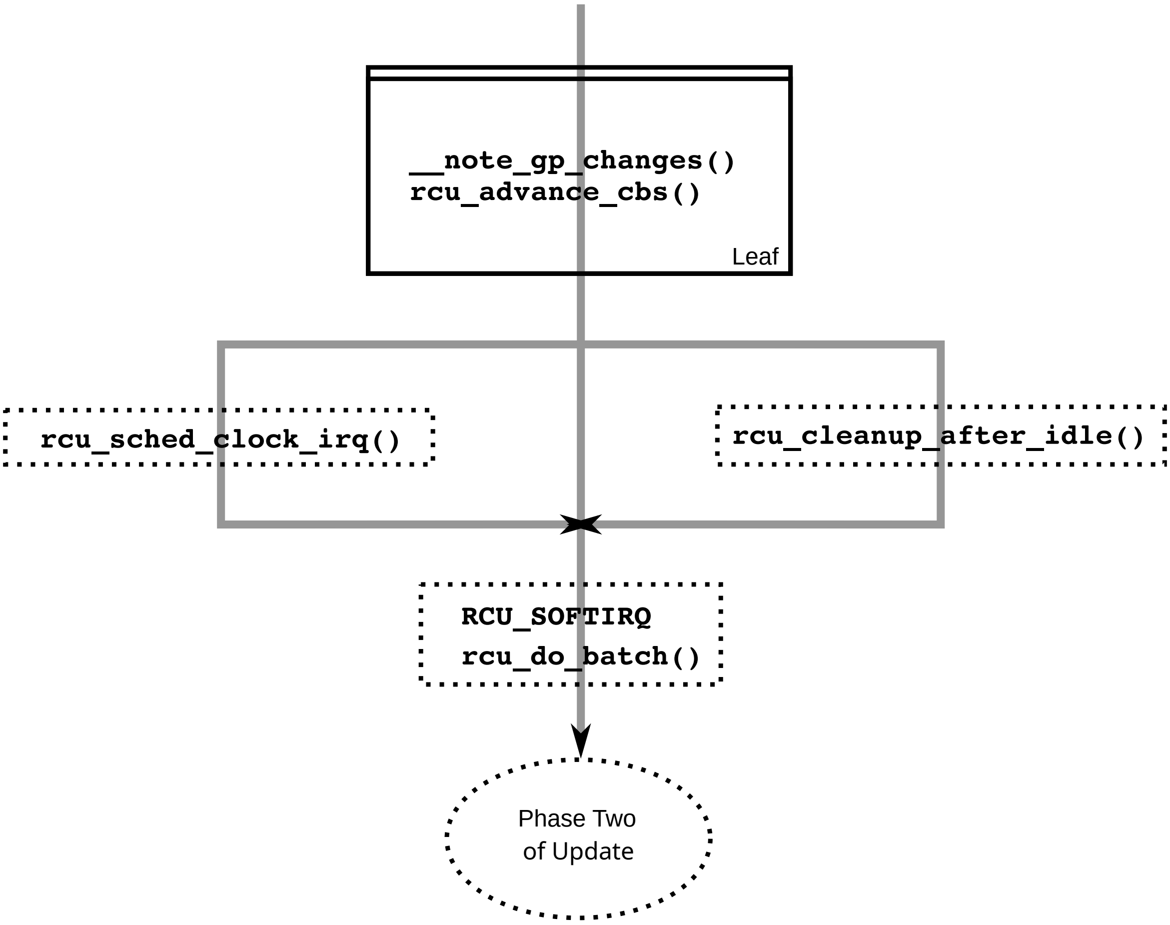
Callback Invocation

Once a given CPU's leaf rcu_node structure's ->gp_seq field has been updated, that CPU can begin invoking its RCU callbacks that were waiting for this grace period to end. These callbacks are identified by rcu_advance_cbs(), which is usually invoked by __note_gp_changes(). As shown in the diagram below, this invocation can be triggered by the scheduling-clock interrupt (rcu_sched_clock_irq() on the left) or by idle entry (rcu_cleanup_after_idle() on the right, but only for kernels build with CONFIG_RCU_FAST_NO_HZ=y). Either way, RCU_SOFTIRQ is raised, which results in rcu_do_batch() invoking the callbacks, which in turn allows those callbacks to carry out (either directly or indirectly via wakeup) the needed phase-two processing for each update.

Please note that callback invocation can also be prompted by any number of corner-case code paths, for example, when a CPU notes that it has excessive numbers of callbacks queued. In all cases, the CPU acquires its leaf rcu_node structure's ->lock before invoking callbacks, which preserves the required ordering against the newly completed grace period.

However, if the callback function communicates to other CPUs, for example, doing a wakeup, then it is that function's responsibility to maintain ordering. For example, if the callback function wakes up a task that runs on some other CPU, proper ordering must in place in both the callback function and the task being awakened. To see why this is important, consider the top half of the *grace-period cleanup* diagram. The callback might be running on a CPU corresponding to the leftmost leaf rcu_node structure, and awaken a task that is to run on a CPU corresponding to the rightmost leaf rcu_node structure, and the grace-period kernel thread might not yet have reached the rightmost leaf. In this case, the grace period's memory ordering might not yet have reached that CPU, so again the callback function and the awakened task must supply proper ordering.


rcu_gp_cleanup()
rcu_seq_end(&rnp->gp_seq)
Root
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
Leaf
Leaf
Leaf
Leaf
rcu_seq_end(&rsp->gp_seq)
Root
Leaf
Leaf
Leaf
Leaf
Start of
Next Grace
Period


__note_gp_changes()
rcu_advance_cbs()
Leaf
rcu_sched_clock_irq()
rcu_cleanup_after_idle()
RCU_SOFTIRQ
rcu_do_batch()
Phase Two
of Update

Putting It All Together

A stitched-together diagram is here:

Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

4.5.16 A Tour Through TREE_RCU's Expedited Grace Periods

Introduction

This document describes RCU's expedited grace periods. Unlike RCU's normal grace periods, which accept long latencies to attain high efficiency and minimal disturbance, expedited grace periods accept lower efficiency and significant disturbance to attain shorter latencies.

There are two flavors of RCU (RCU-preempt and RCU-sched), with an earlier third RCU-bh flavor having been implemented in terms of the other two. Each of the two implementations is covered in its own section.

Expedited Grace Period Design

The expedited RCU grace periods cannot be accused of being subtle, given that they for all intents and purposes hammer every CPU that has not yet provided a quiescent state for the current expedited grace period. The one saving grace is that the hammer has grown a bit smaller over time: The old call to `try_stop_cpus()` has been replaced with a set of calls to `smp_call_function_single()`, each of which results in an IPI to the target CPU. The corresponding handler function checks the CPU's state, motivating a faster quiescent state where possible, and triggering a report of that quiescent state. As always for RCU, once everything has spent some time in a quiescent state, the expedited grace period has completed.

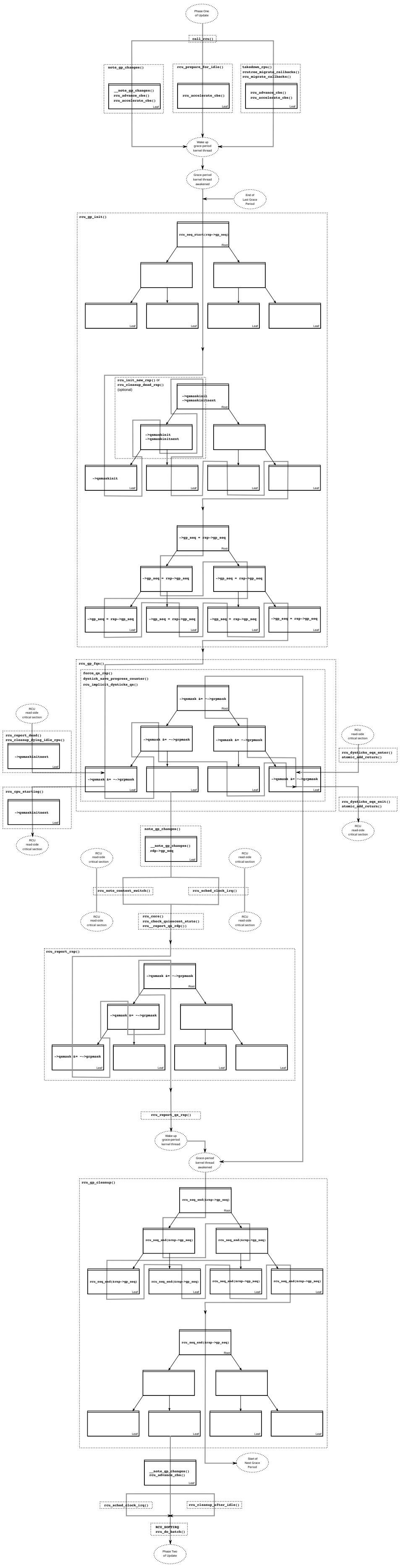
The details of the `smp_call_function_single()` handler's operation depend on the RCU flavor, as described in the following sections.

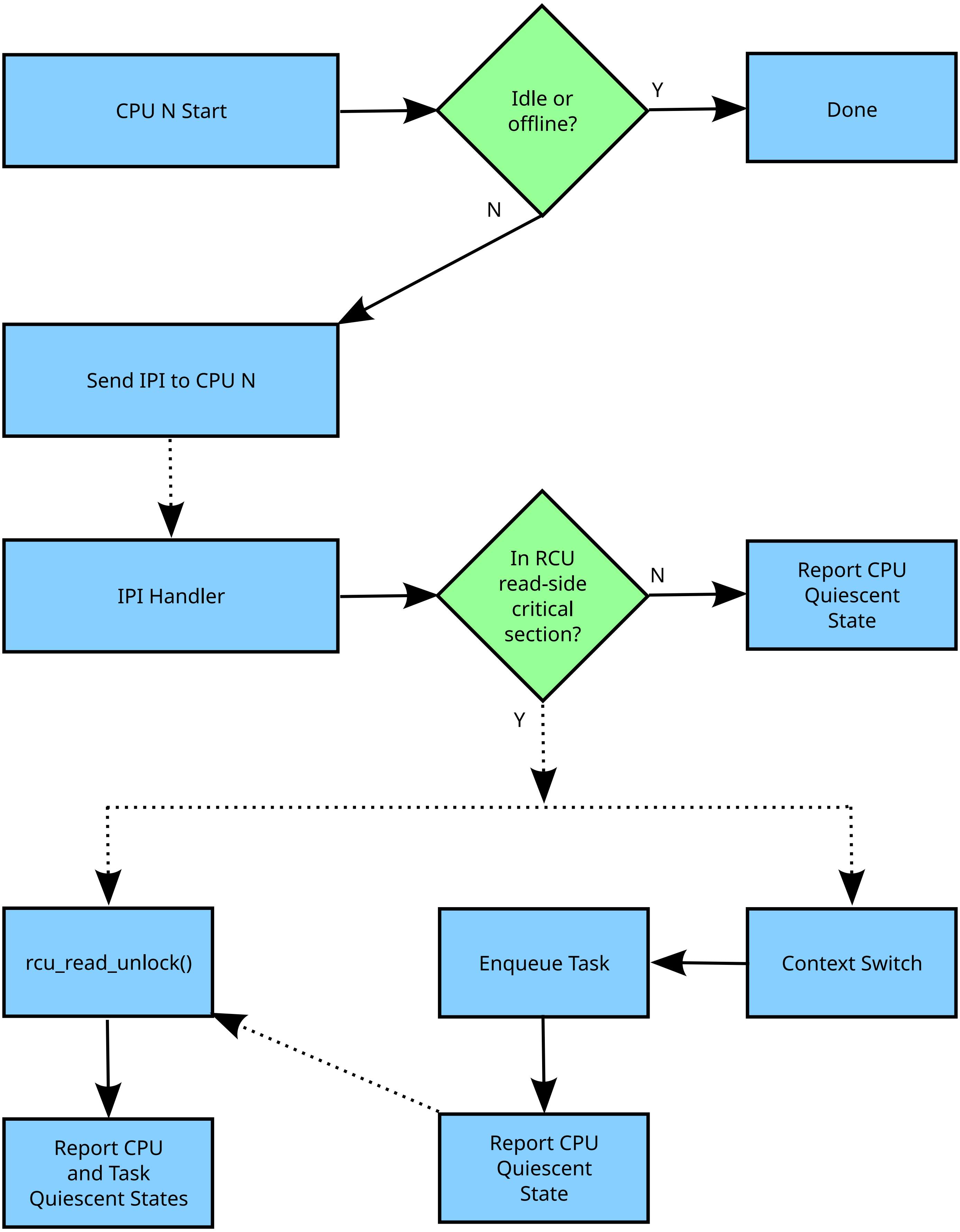
RCU-preempt Expedited Grace Periods

`CONFIG_PREEMPTION=y` kernels implement RCU-preempt. The overall flow of the handling of a given CPU by an RCU-preempt expedited grace period is shown in the following diagram:

The solid arrows denote direct action, for example, a function call. The dotted arrows denote indirect action, for example, an IPI or a state that is reached after some time.

If a given CPU is offline or idle, `synchronize_rcu_expedited()` will ignore it because idle and offline CPUs are already residing in quiescent states. Otherwise, the expedited grace period will use `smp_call_function_single()` to send the CPU an IPI, which is handled by `rcu_exp_handler()`.




CPU N Start
Idle or offline?
Y
Done
N
Send IPI to CPU N
IPI Handler
In RCU read-side critical section?
N
Report CPU Quiescent State
Y
rcu_read_unlock()
Enqueue Task
Context Switch
Report CPU and Task Quiescent States
Report CPU Quiescent State

However, because this is preemptible RCU, `rcu_exp_handler()` can check to see if the CPU is currently running in an RCU read-side critical section. If not, the handler can immediately report a quiescent state. Otherwise, it sets flags so that the outermost `rcu_read_unlock()` invocation will provide the needed quiescent-state report. This flag-setting avoids the previous forced preemption of all CPUs that might have RCU read-side critical sections. In addition, this flag-setting is done so as to avoid increasing the overhead of the common-case fastpath through the scheduler.

Again because this is preemptible RCU, an RCU read-side critical section can be preempted. When that happens, RCU will enqueue the task, which will the continue to block the current expedited grace period until it resumes and finds its outermost `rcu_read_unlock()`. The CPU will report a quiescent state just after enqueuing the task because the CPU is no longer blocking the grace period. It is instead the preempted task doing the blocking. The list of blocked tasks is managed by `rcu_preempt_ctxt_queue()`, which is called from `rcu_preempt_note_context_switch()`, which in turn is called from `rcu_note_context_switch()`, which in turn is called from the scheduler.

Quick Quiz:
Why not just have the expedited grace period check the state of all the CPUs? After all, that would avoid all those real-time-unfriendly IPIs.
Answer:
Because we want the RCU read-side critical sections to run fast, which means no memory barriers. Therefore, it is not possible to safely check the state from some other CPU. And even if it was possible to safely check the state, it would still be necessary to IPI the CPU to safely interact with the upcoming `rcu_read_unlock()` invocation, which means that the remote state testing would not help the worst-case latency that real-time applications care about. One way to prevent your real-time application from getting hit with these IPIs is to build your kernel with `CONFIG_NO_HZ_FULL=y`. RCU would then perceive the CPU running your application as being idle, and it would be able to safely detect that state without needing to IPI the CPU.

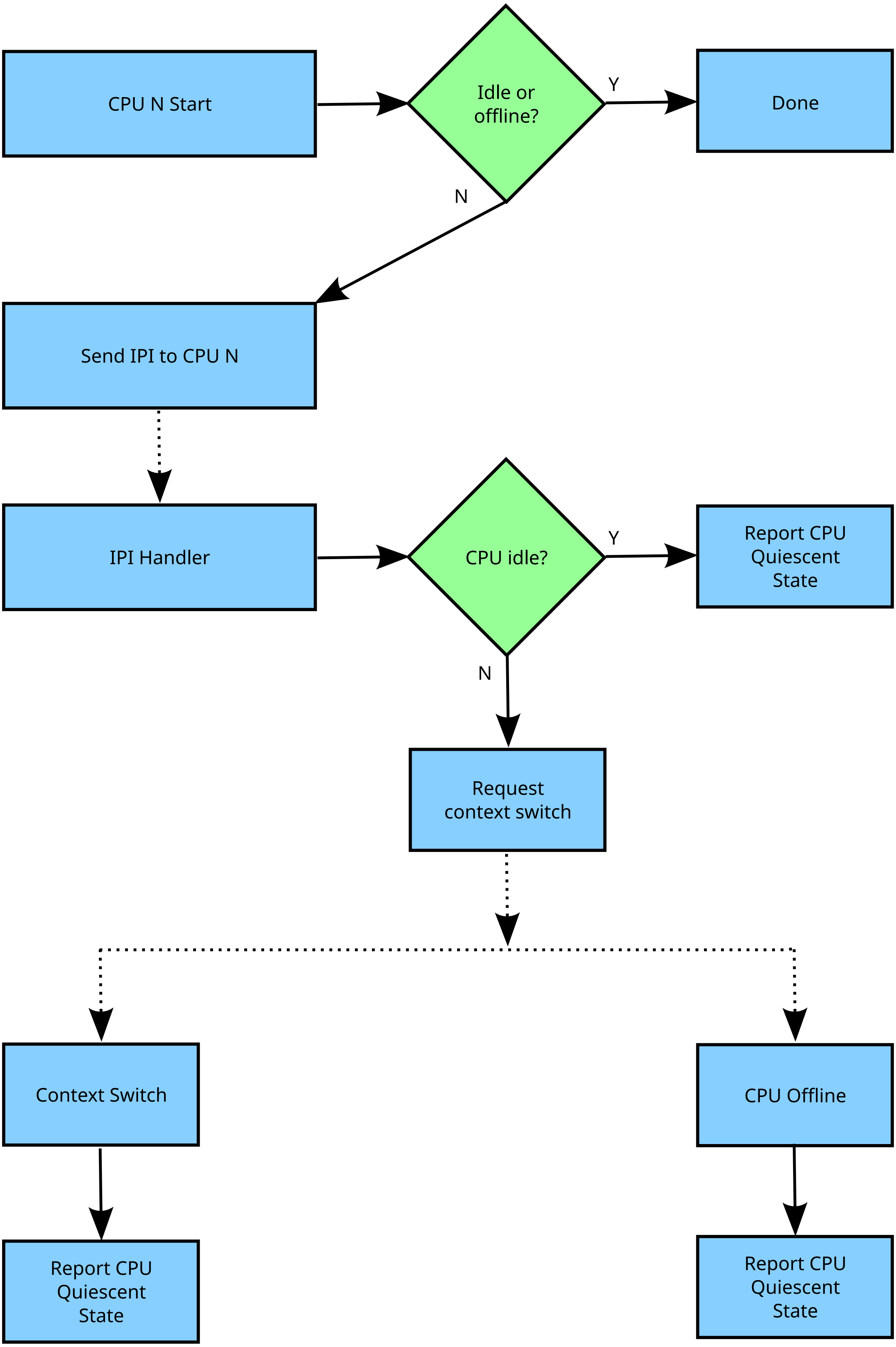
Please note that this is just the overall flow: Additional complications can arise due to races with CPUs going idle or offline, among other things.

RCU-sched Expedited Grace Periods

`CONFIG_PREEMPTION=n` kernels implement RCU-sched. The overall flow of the handling of a given CPU by an RCU-sched expedited grace period is shown in the following diagram:

As with RCU-preempt, RCU-sched's `synchronize_rcu_expedited()` ignores offline and idle CPUs, again because they are in remotely detectable quiescent states. However, because the `rcu_read_lock_sched()` and `rcu_read_unlock_sched()` leave no trace of their invocation, in general it is not possible to tell whether or not the current CPU is in an RCU read-side critical section. The best that RCU-sched's `rcu_exp_handler()` can do is to check for idle, on the off-chance that the CPU went idle while the IPI was in flight. If the CPU is idle, then `rcu_exp_handler()` reports the quiescent state.

Otherwise, the handler forces a future context switch by setting the NEED_RESCHED flag of the current task's thread flag and the CPU preempt counter. At the time of the context switch,


CPU N Start
Idle or offline?
Y
Done
N
Send IPI to CPU N
IPI Handler
CPU idle?
Y
Report CPU Quiescent State
N
Request context switch
Context Switch
CPU Offline
Report CPU Quiescent State
Report CPU Quiescent State

the CPU reports the quiescent state. Should the CPU go offline first, it will report the quiescent state at that time.

Expedited Grace Period and CPU Hotplug

The expedited nature of expedited grace periods require a much tighter interaction with CPU hotplug operations than is required for normal grace periods. In addition, attempting to IPI offline CPUs will result in splats, but failing to IPI online CPUs can result in too-short grace periods. Neither option is acceptable in production kernels.

The interaction between expedited grace periods and CPU hotplug operations is carried out at several levels:

1. The number of CPUs that have ever been online is tracked by the `rcu_state` structure's `->ncpus` field. The `rcu_state` structure's `->ncpus_snap` field tracks the number of CPUs that have ever been online at the beginning of an RCU expedited grace period. Note that this number never decreases, at least in the absence of a time machine.
2. The identities of the CPUs that have ever been online is tracked by the `rcu_node` structure's `->expmaskinitnext` field. The `rcu_node` structure's `->expmaskinit` field tracks the identities of the CPUs that were online at least once at the beginning of the most recent RCU expedited grace period. The `rcu_state` structure's `->ncpus` and `->ncpus_snap` fields are used to detect when new CPUs have come online for the first time, that is, when the `rcu_node` structure's `->expmaskinitnext` field has changed since the beginning of the last RCU expedited grace period, which triggers an update of each `rcu_node` structure's `->expmaskinit` field from its `->expmaskinitnext` field.
3. Each `rcu_node` structure's `->expmaskinit` field is used to initialize that structure's `->expmask` at the beginning of each RCU expedited grace period. This means that only those CPUs that have been online at least once will be considered for a given grace period.
4. Any CPU that goes offline will clear its bit in its leaf `rcu_node` structure's `->qsmaskinitnext` field, so any CPU with that bit clear can safely be ignored. However, it is possible for a CPU coming online or going offline to have this bit set for some time while `cpu_online` returns `false`.
5. For each non-idle CPU that RCU believes is currently online, the grace period invokes `smp_call_function_single()`. If this succeeds, the CPU was fully online. Failure indicates that the CPU is in the process of coming online or going offline, in which case it is necessary to wait for a short time period and try again. The purpose of this wait (or series of waits, as the case may be) is to permit a concurrent CPU-hotplug operation to complete.
6. In the case of RCU-sched, one of the last acts of an outgoing CPU is to invoke `rcu_report_dead()`, which reports a quiescent state for that CPU. However, this is likely paranoia-induced redundancy.

Quick Quiz:

Why all the dancing around with multiple counters and masks tracking CPUs that were once online? Why not just have a single set of masks tracking the currently online CPUs and be done with it?

Answer:

Maintaining single set of masks tracking the online CPUs *sounds* easier, at least until you try working out all the race conditions between grace-period initialization and CPU-hotplug operations. For example, suppose initialization is progressing down the tree while a CPU-offline operation is progressing up the tree. This situation can result in bits set at the top of the tree that have no counterparts at the bottom of the tree. Those bits will never be cleared, which will result in grace-period hangs. In short, that way lies madness, to say nothing of a great many bugs, hangs, and deadlocks. In contrast, the current multi-mask multi-counter scheme ensures that grace-period initialization will always see consistent masks up and down the tree, which brings significant simplifications over the single-mask method.

This is an instance of deferring work in order to avoid synchronization. Lazily recording CPU-hotplug events at the beginning of the next grace period greatly simplifies maintenance of the CPU-tracking bitmasks in the `rcu_node` tree.

Expedited Grace Period Refinements

Idle-CPU Checks

Each expedited grace period checks for idle CPUs when initially forming the mask of CPUs to be IPIed and again just before IPIing a CPU (both checks are carried out by `sync_rcu_exp_select_cpus()`). If the CPU is idle at any time between those two times, the CPU will not be IPIed. Instead, the task pushing the grace period forward will include the idle CPUs in the mask passed to `rcu_report_exp_cpu_mult()`.

For RCU-sched, there is an additional check: If the IPI has interrupted the idle loop, then `rcu_exp_handler()` invokes `rcu_report_exp_rdp()` to report the corresponding quiescent state.

For RCU-preempt, there is no specific check for idle in the IPI handler (`rcu_exp_handler()`), but because RCU read-side critical sections are not permitted within the idle loop, if `rcu_exp_handler()` sees that the CPU is within RCU read-side critical section, the CPU cannot possibly be idle. Otherwise, `rcu_exp_handler()` invokes `rcu_report_exp_rdp()` to report the corresponding quiescent state, regardless of whether or not that quiescent state was due to the CPU being idle.

In summary, RCU expedited grace periods check for idle when building the bitmask of CPUs that must be IPIed, just before sending each IPI, and (either explicitly or implicitly) within the IPI handler.

Batching via Sequence Counter

If each grace-period request was carried out separately, expedited grace periods would have abysmal scalability and problematic high-load characteristics. Because each grace-period operation can serve an unlimited number of updates, it is important to *batch* requests, so that a single expedited grace-period operation will cover all requests in the corresponding batch.

This batching is controlled by a sequence counter named `->expedited_sequence` in the `rcu_state` structure. This counter has an odd value when there is an expedited grace period in progress and an even value otherwise, so that dividing the counter value by two gives the number of completed grace periods. During any given update request, the counter must transition from even to odd and then back to even, thus indicating that a grace period has elapsed. Therefore, if the initial value of the counter is `s`, the updater must wait until the counter reaches at least the value `(s+3)&~0x1`. This counter is managed by the following access functions:

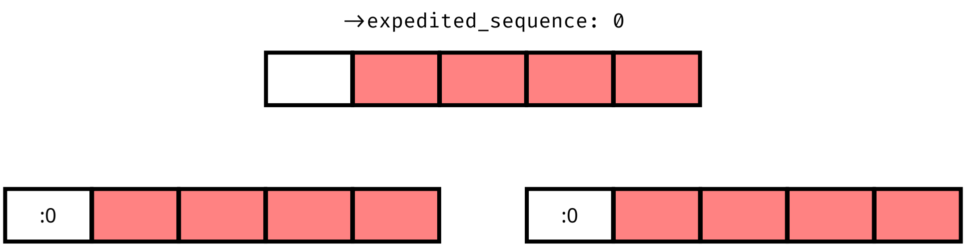
1. `rcu_exp_gp_seq_start()`, which marks the start of an expedited grace period.
2. `rcu_exp_gp_seq_end()`, which marks the end of an expedited grace period.
3. `rcu_exp_gp_seq_snap()`, which obtains a snapshot of the counter.
4. `rcu_exp_gp_seq_done()`, which returns `true` if a full expedited grace period has elapsed since the corresponding call to `rcu_exp_gp_seq_snap()`.

Again, only one request in a given batch need actually carry out a grace-period operation, which means there must be an efficient way to identify which of many concurrent requests will initiate the grace period, and that there be an efficient way for the remaining requests to wait for that grace period to complete. However, that is the topic of the next section.

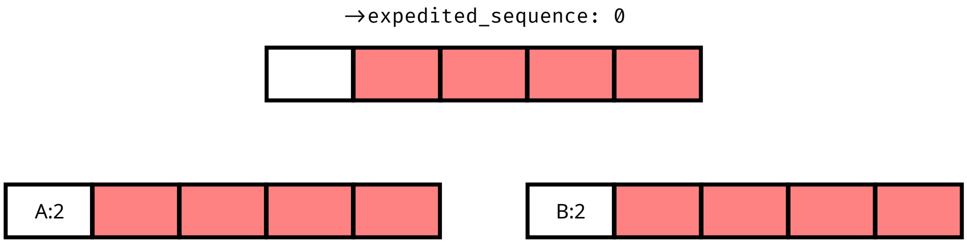
Funnel Locking and Wait/Wakeup

The natural way to sort out which of a batch of updaters will initiate the expedited grace period is to use the `rcu_node` combining tree, as implemented by the `exp_funnel_lock()` function. The first updater corresponding to a given grace period arriving at a given `rcu_node` structure records its desired grace-period sequence number in the `->exp_seq_rq` field and moves up to the next level in the tree. Otherwise, if the `->exp_seq_rq` field already contains the sequence number for the desired grace period or some later one, the updater blocks on one of four wait queues in the `->exp_wq[]` array, using the second-from-bottom and third-from bottom bits as an index. An `->exp_lock` field in the `rcu_node` structure synchronizes access to these fields.

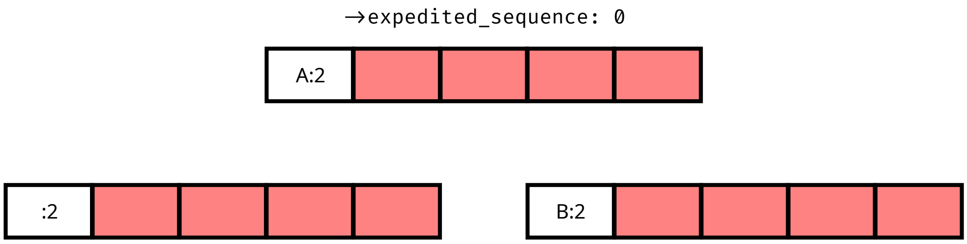
An empty `rcu_node` tree is shown in the following diagram, with the white cells representing the `->exp_seq_rq` field and the red cells representing the elements of the `->exp_wq[]` array.



The next diagram shows the situation after the arrival of Task A and Task B at the leftmost and rightmost leaf `rcu_node` structures, respectively. The current value of the `rcu_state` structure's `->expedited_sequence` field is zero, so adding three and clearing the bottom bit results in the value two, which both tasks record in the ->exp_seq_rq field of their respective rcu_node structures:



Each of Tasks A and B will move up to the root rcu_node structure. Suppose that Task A wins, recording its desired grace-period sequence number and resulting in the state shown below:



Task A now advances to initiate a new grace period, while Task B moves up to the root rcu_node structure, and, seeing that its desired sequence number is already recorded, blocks on ->exp_wq[1].

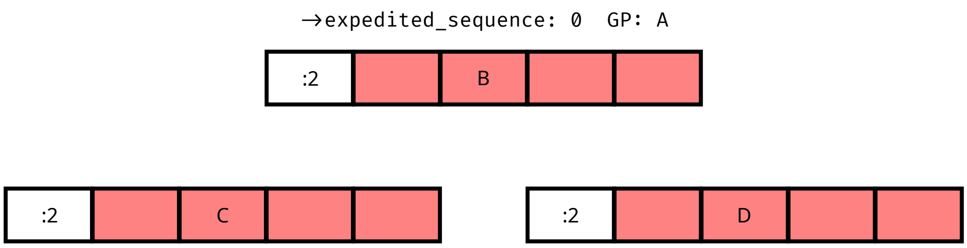
Quick Quiz:

Why ->exp_wq[1]? Given that the value of these tasks' desired sequence number is two, so shouldn't they instead block on ->exp_wq[2]?

Answer:

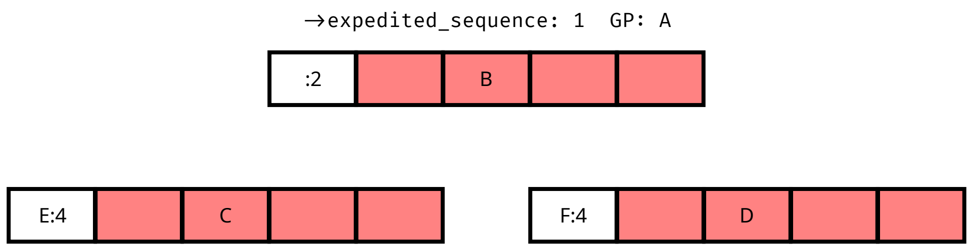
No. Recall that the bottom bit of the desired sequence number indicates whether or not a grace period is currently in progress. It is therefore necessary to shift the sequence number right one bit position to obtain the number of the grace period. This results in ->exp_wq[1].

If Tasks C and D also arrive at this point, they will compute the same desired grace-period sequence number, and see that both leaf rcu_node structures already have that value recorded. They will therefore block on their respective rcu_node structures' ->exp_wq[1] fields, as shown below:

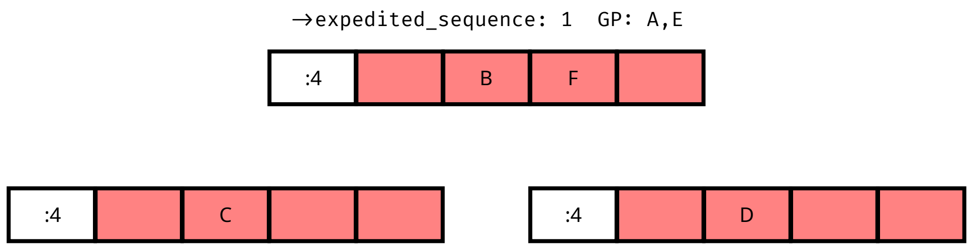


Task A now acquires the rcu_state structure's ->exp_mutex and initiates the grace period, which increments ->expedited_sequence. Therefore, if Tasks E and F arrive, they will compute a desired sequence number of 4 and will record this value as shown below:

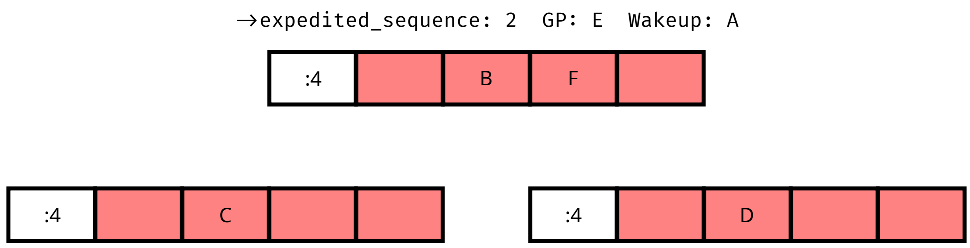
Tasks E and F will propagate up the rcu_node combining tree, with Task F blocking on the root



rcu_node structure and Task E wait for Task A to finish so that it can start the next grace period. The resulting state is as shown below:



Once the grace period completes, Task A starts waking up the tasks waiting for this grace period to complete, increments the ->expedited_sequence, acquires the ->exp_wake_mutex and then releases the ->exp_mutex. This results in the following state:

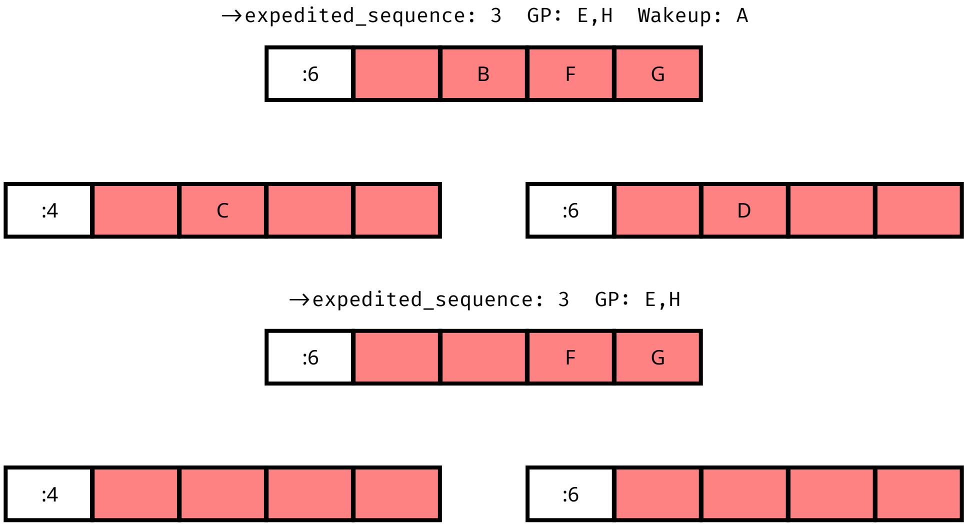


Task E can then acquire ->exp_mutex and increment ->expedited_sequence to the value three. If new tasks G and H arrive and moves up the combining tree at the same time, the state will be as follows:

Note that three of the root rcu_node structure's waitqueues are now occupied. However, at some point, Task A will wake up the tasks blocked on the ->exp_wq waitqueues, resulting in the following state:

Execution will continue with Tasks E and H completing their grace periods and carrying out their wakeups.

Quick Quiz:
What happens if Task A takes so long to do its wakeups that Task E's grace period completes?
Answer:
Then Task E will block on the ->exp_wake_mutex, which will also prevent it from releasing ->exp_mutex, which in turn will prevent the next grace period from starting. This last is important in preventing overflow of the ->exp_wq[] array.



Use of Workqueues

In earlier implementations, the task requesting the expedited grace period also drove it to completion. This straightforward approach had the disadvantage of needing to account for POSIX signals sent to user tasks, so more recent implementations use the Linux kernel's workqueues (see *Workqueue*).

The requesting task still does counter snapshotting and funnel-lock processing, but the task reaching the top of the funnel lock does a `schedule_work()` (from `_synchronize_rcu_expedited()` so that a workqueue kthread does the actual grace-period processing. Because workqueue kthreads do not accept POSIX signals, grace-period-wait processing need not allow for POSIX signals. In addition, this approach allows wakeups for the previous expedited grace period to be overlapped with processing for the next expedited grace period. Because there are only four sets of waitqueues, it is necessary to ensure that the previous grace period's wakeups complete before the next grace period's wakeups start. This is handled by having the `->exp_mutex` guard expedited grace-period processing and the `->exp_wake_mutex` guard wakeups. The key point is that the `->exp_mutex` is not released until the first wakeup is complete, which means that the `->exp_wake_mutex` has already been acquired at that point. This approach ensures that the previous grace period's wakeups can be carried out while the current grace period is in process, but that these wakeups will complete before the next grace period starts. This means that only three waitqueues are required, guaranteeing that the four that are provided are sufficient.

Stall Warnings

Expediting grace periods does nothing to speed things up when RCU readers take too long, and therefore expedited grace periods check for stalls just as normal grace periods do.

Quick Quiz:

But why not just let the normal grace-period machinery detect the stalls, given that a given reader must block both normal and expedited grace periods?

Answer:

Because it is quite possible that at a given time there is no normal grace period in progress, in which case the normal grace period cannot emit a stall warning.

The `synchronize_sched_expedited_wait()` function loops waiting for the expedited grace period to end, but with a timeout set to the current RCU CPU stall-warning time. If this time is exceeded, any CPUs or `rcu_node` structures blocking the current grace period are printed. Each stall warning results in another pass through the loop, but the second and subsequent passes use longer stall times.

Mid-boot operation

The use of workqueues has the advantage that the expedited grace-period code need not worry about POSIX signals. Unfortunately, it has the corresponding disadvantage that workqueues cannot be used until they are initialized, which does not happen until some time after the scheduler spawns the first task. Given that there are parts of the kernel that really do want to execute grace periods during this mid-boot "dead zone", expedited grace periods must do something else during this time.

What they do is to fall back to the old practice of requiring that the requesting task drive the expedited grace period, as was the case before the use of workqueues. However, the requesting task is only required to drive the grace period during the mid-boot dead zone. Before mid-boot, a synchronous grace period is a no-op. Some time after mid-boot, workqueues are used.

Non-expedited non-SRCU synchronous grace periods must also operate normally during mid-boot. This is handled by causing non-expedited grace periods to take the expedited code path during mid-boot.

The current code assumes that there are no POSIX signals during the mid-boot dead zone. However, if an overwhelming need for POSIX signals somehow arises, appropriate adjustments can be made to the expedited stall-warning code. One such adjustment would reinstate the pre-workqueue stall-warning checks, but only during the mid-boot dead zone.

With this refinement, synchronous grace periods can now be used from task context pretty much any time during the life of the kernel. That is, aside from some points in the suspend, hibernate, or shutdown code path.

Summary

Expedited grace periods use a sequence-number approach to promote batching, so that a single grace-period operation can serve numerous requests. A funnel lock is used to efficiently identify the one task out of a concurrent group that will request the grace period. All members of the group will block on waitqueues provided in the `rcu_node` structure. The actual grace-period processing is carried out by a workqueue.

CPU-hotplug operations are noted lazily in order to prevent the need for tight synchronization between expedited grace periods and CPU-hotplug operations. The dyntick-idle counters are used to avoid sending IPIs to idle CPUs, at least in the common case. RCU-preempt and RCU-sched use different IPI handlers and different code to respond to the state changes carried out by those handlers, but otherwise use common code.

Quiescent states are tracked using the `rcu_node` tree, and once all necessary quiescent states have been reported, all tasks waiting on this expedited grace period are awakened. A pair of mutexes are used to allow one grace period's wakeups to proceed concurrently with the next grace period's processing.

This combination of mechanisms allows expedited grace periods to run reasonably efficiently. However, for non-time-critical tasks, normal grace periods should be used instead because their longer duration permits much higher degrees of batching, and thus much lower per-request overheads.

4.5.17 A Tour Through RCU's Requirements

Author: Paul E. McKenney

The initial version of this document appeared in the LWN on those articles: part 1, part 2, and part 3.

Introduction

Read-copy update (RCU) is a synchronization mechanism that is often used as a replacement for reader-writer locking. RCU is unusual in that updaters do not block readers, which means that RCU's read-side primitives can be exceedingly fast and scalable. In addition, updaters can make useful forward progress concurrently with readers. However, all this concurrency between RCU readers and updaters does raise the question of exactly what RCU readers are doing, which in turn raises the question of exactly what RCU's requirements are.

This document therefore summarizes RCU's requirements, and can be thought of as an informal, high-level specification for RCU. It is important to understand that RCU's specification is primarily empirical in nature; in fact, I learned about many of these requirements the hard way. This situation might cause some consternation, however, not only has this learning process been a lot of fun, but it has also been a great privilege to work with so many people willing to apply technologies in interesting new ways.

All that aside, here are the categories of currently known RCU requirements:

1. *Fundamental Requirements*
2. *Fundamental Non-Requirements*
3. *Parallelism Facts of Life*
4. *Quality-of-Implementation Requirements*
5. *Linux Kernel Complications*
6. *Software-Engineering Requirements*
7. *Other RCU Flavors*
8. *Possible Future Changes*

This is followed by a *summary*, however, the answers to each quick quiz immediately follows the quiz. Select the big white space with your mouse to see the answer.

Fundamental Requirements

RCU's fundamental requirements are the closest thing RCU has to hard mathematical requirements. These are:

1. *Grace-Period Guarantee*
2. *Publish/Subscribe Guarantee*
3. *Memory-Barrier Guarantees*
4. *RCU Primitives Guaranteed to Execute Unconditionally*
5. *Guaranteed Read-to-Write Upgrade*

Grace-Period Guarantee

RCU's grace-period guarantee is unusual in being premeditated: Jack Slingwine and I had this guarantee firmly in mind when we started work on RCU (then called "rclock") in the early 1990s. That said, the past two decades of experience with RCU have produced a much more detailed understanding of this guarantee.

RCU's grace-period guarantee allows updaters to wait for the completion of all pre-existing RCU read-side critical sections. An RCU read-side critical section begins with the marker rcu_read_lock() and ends with the marker rcu_read_unlock(). These markers may be nested, and RCU treats a nested set as one big RCU read-side critical section. Production-quality implementations of rcu_read_lock() and rcu_read_unlock() are extremely lightweight, and in fact have exactly zero overhead in Linux kernels built for production use with `CONFIG_PREEMPTION=n`.

This guarantee allows ordering to be enforced with extremely low overhead to readers, for example:

```
 1 int x, y;
 2
 3 void thread0(void)
 4 {
 5   rcu_read_lock();
 6   r1 = READ_ONCE(x);
 7   r2 = READ_ONCE(y);
 8   rcu_read_unlock();
 9 }
10
11 void thread1(void)
12 {
13   WRITE_ONCE(x, 1);
14   synchronize_rcu();
15   WRITE_ONCE(y, 1);
16 }
```

Because the synchronize_rcu() on line 14 waits for all pre-existing readers, any instance of thread0() that loads a value of zero from `x` must complete before thread1() stores to `y`, so that instance must also load a value of zero from `y`. Similarly, any instance of thread0() that loads a value of one from `y` must have started after the synchronize_rcu() started, and must therefore also load a value of one from `x`. Therefore, the outcome:

```
(r1 == 0 && r2 == 1)
```

cannot happen.

Quick Quiz:

Wait a minute! You said that updaters can make useful forward progress concurrently with readers, but pre-existing readers will block synchronize_rcu()!!! Just who are you trying to fool???

Answer:

First, if updaters do not wish to be blocked by readers, they can use call_rcu() or kfree_rcu(), which will be discussed later. Second, even when using synchronize_rcu(), the other update-side code does run concurrently with readers, whether pre-existing or not.

This scenario resembles one of the first uses of RCU in DYNIX/ptx, which managed a distributed lock manager's transition into a state suitable for handling recovery from node failure, more or less as follows:

```
#define STATE_NORMAL        0
#define STATE_WANT_RECOVERY 1
#define STATE_RECOVERING    2
#define STATE_WANT_NORMAL   3

int state = STATE_NORMAL;

void do_something_dlm(void)
{
  int state_snap;

  rcu_read_lock();
  state_snap = READ_ONCE(state);
  if (state_snap == STATE_NORMAL)
    do_something();
  else
    do_something_carefully();
  rcu_read_unlock();
}

void start_recovery(void)
{
  WRITE_ONCE(state, STATE_WANT_RECOVERY);
  synchronize_rcu();
  WRITE_ONCE(state, STATE_RECOVERING);
  recovery();
  WRITE_ONCE(state, STATE_WANT_NORMAL);
  synchronize_rcu();
  WRITE_ONCE(state, STATE_NORMAL);
}
```

The RCU read-side critical section in do_something_dlm() works with the synchronize_rcu() in start_recovery() to guarantee that do_something() never runs concurrently with recovery(), but with little or no synchronization overhead in do_something_dlm().

Quick Quiz:
Why is the synchronize_rcu() on line 28 needed?
Answer:
Without that extra grace period, memory reordering could result in do_something_dlm() executing do_something() concurrently with the last bits of recovery().

In order to avoid fatal problems such as deadlocks, an RCU read-side critical section must not contain calls to synchronize_rcu(). Similarly, an RCU read-side critical section must not contain anything that waits, directly or indirectly, on completion of an invocation of synchronize_rcu().

Although RCU's grace-period guarantee is useful in and of itself, with quite a few use cases, it would be good to be able to use RCU to coordinate read-side access to linked data structures. For this, the grace-period guarantee is not sufficient, as can be seen in function add_gp_buggy() below. We will look at the reader's code later, but in the meantime, just think of the reader as locklessly picking up the `gp` pointer, and, if the value loaded is non-`NULL`, locklessly accessing the `->a` and `->b` fields.

```
bool add_gp_buggy(int a, int b)
{
  p = kmalloc(sizeof(*p), GFP_KERNEL);
  if (!p)
    return -ENOMEM;
  spin_lock(&gp_lock);
  if (rcu_access_pointer(gp)) {
    spin_unlock(&gp_lock);
    return false;
  }
  p->a = a;
  p->b = a;
  gp = p; /* ORDERING BUG */
  spin_unlock(&gp_lock);
  return true;
}
```

The problem is that both the compiler and weakly ordered CPUs are within their rights to reorder this code as follows:

```
bool add_gp_buggy_optimized(int a, int b)
{
  p = kmalloc(sizeof(*p), GFP_KERNEL);
  if (!p)
    return -ENOMEM;
  spin_lock(&gp_lock);
  if (rcu_access_pointer(gp)) {
    spin_unlock(&gp_lock);
    return false;
  }
  gp = p; /* ORDERING BUG */
  p->a = a;
```

```
13   p->b = a;
14   spin_unlock(&gp_lock);
15   return true;
16 }
```

If an RCU reader fetches `gp` just after `add_gp_buggy_optimized` executes line 11, it will see garbage in the `->a` and `->b` fields. And this is but one of many ways in which compiler and hardware optimizations could cause trouble. Therefore, we clearly need some way to prevent the compiler and the CPU from reordering in this manner, which brings us to the publish-subscribe guarantee discussed in the next section.

Publish/Subscribe Guarantee

RCU's publish-subscribe guarantee allows data to be inserted into a linked data structure without disrupting RCU readers. The updater uses rcu_assign_pointer() to insert the new data, and readers use rcu_dereference() to access data, whether new or old. The following shows an example of insertion:

```
 1 bool add_gp(int a, int b)
 2 {
 3   p = kmalloc(sizeof(*p), GFP_KERNEL);
 4   if (!p)
 5     return -ENOMEM;
 6   spin_lock(&gp_lock);
 7   if (rcu_access_pointer(gp)) {
 8     spin_unlock(&gp_lock);
 9     return false;
10   }
11   p->a = a;
12   p->b = a;
13   rcu_assign_pointer(gp, p);
14   spin_unlock(&gp_lock);
15   return true;
16 }
```

The rcu_assign_pointer() on line 13 is conceptually equivalent to a simple assignment statement, but also guarantees that its assignment will happen after the two assignments in lines 11 and 12, similar to the C11 `memory_order_release` store operation. It also prevents any number of "interesting" compiler optimizations, for example, the use of `gp` as a scratch location immediately preceding the assignment.

Quick Quiz:

But rcu_assign_pointer() does nothing to prevent the two assignments to `p->a` and `p->b` from being reordered. Can't that also cause problems?

Answer:

No, it cannot. The readers cannot see either of these two fields until the assignment to `gp`, by which time both fields are fully initialized. So reordering the assignments to `p->a` and `p->b` cannot possibly cause any problems.

It is tempting to assume that the reader need not do anything special to control its accesses to the RCU-protected data, as shown in do_something_gp_buggy() below:

```
bool do_something_gp_buggy(void)
{
  rcu_read_lock();
  p = gp;  /* OPTIMIZATIONS GALORE!!! */
  if (p) {
    do_something(p->a, p->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

However, this temptation must be resisted because there are a surprisingly large number of ways that the compiler (or weak ordering CPUs like the DEC Alpha) can trip this code up. For but one example, if the compiler were short of registers, it might choose to refetch from gp rather than keeping a separate copy in p as follows:

```
bool do_something_gp_buggy_optimized(void)
{
  rcu_read_lock();
  if (gp) { /* OPTIMIZATIONS GALORE!!! */
    do_something(gp->a, gp->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

If this function ran concurrently with a series of updates that replaced the current structure with a new one, the fetches of gp->a and gp->b might well come from two different structures, which could cause serious confusion. To prevent this (and much else besides), do_something_gp() uses rcu_dereference() to fetch from gp:

```
bool do_something_gp(void)
{
  rcu_read_lock();
  p = rcu_dereference(gp);
  if (p) {
    do_something(p->a, p->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

The rcu_dereference() uses volatile casts and (for DEC Alpha) memory barriers in the Linux kernel. Should a high-quality implementation of C11 `memory_order_consume` [PDF] ever appear, then rcu_dereference() could be implemented as a `memory_order_consume` load. Regardless of the exact implementation, a pointer fetched by rcu_dereference() may not be used outside of the outermost RCU read-side critical section containing that rcu_dereference(), unless protection of the corresponding data element has been passed from RCU to some other synchronization mechanism, most commonly locking or reference counting (see ../../rcuref.rst).

In short, updaters use rcu_assign_pointer() and readers use rcu_dereference(), and these two RCU API elements work together to ensure that readers have a consistent view of newly added data elements.

Of course, it is also necessary to remove elements from RCU-protected data structures, for example, using the following process:

1. Remove the data element from the enclosing structure.
2. Wait for all pre-existing RCU read-side critical sections to complete (because only pre-existing readers can possibly have a reference to the newly removed data element).
3. At this point, only the updater has a reference to the newly removed data element, so it can safely reclaim the data element, for example, by passing it to kfree().

This process is implemented by remove_gp_synchronous():

```
 1 bool remove_gp_synchronous(void)
 2 {
 3   struct foo *p;
 4
 5   spin_lock(&gp_lock);
 6   p = rcu_access_pointer(gp);
 7   if (!p) {
 8     spin_unlock(&gp_lock);
 9     return false;
10   }
11   rcu_assign_pointer(gp, NULL);
12   spin_unlock(&gp_lock);
13   synchronize_rcu();
14   kfree(p);
15   return true;
16 }
```

This function is straightforward, with line 13 waiting for a grace period before line 14 frees the old data element. This waiting ensures that readers will reach line 7 of do_something_gp() before the data element referenced by `p` is freed. The *`rcu_access_pointer()`* on line 6 is similar to rcu_dereference(), except that:

1. The value returned by *`rcu_access_pointer()`* cannot be dereferenced. If you want to access the value pointed to as well as the pointer itself, use rcu_dereference() instead of *`rcu_access_pointer()`*.
2. The call to *`rcu_access_pointer()`* need not be protected. In contrast, rcu_dereference() must either be within an RCU read-side critical section or in a code segment where the pointer cannot change, for example, in code protected by the corresponding update-side lock.

Quick Quiz:
Without the rcu_dereference() or the *rcu_access_pointer()*, what destructive optimizations might the compiler make use of?
Answer:
Let's start with what happens to do_something_gp() if it fails to use rcu_dereference(). It could reuse a value formerly fetched from this same pointer. It could also fetch the pointer from `gp` in a byte-at-a-time manner, resulting in *load tearing*, in turn resulting a bytewise mash-up of two distinct pointer values. It might even use value-speculation optimizations, where it makes a wrong guess, but by the time it gets around to checking the value, an update has changed the pointer to match the wrong guess. Too bad about any dereferences that returned pre-initialization garbage in the meantime! For remove_gp_synchronous(), as long as all modifications to `gp` are carried out while holding `gp_lock`, the above optimizations are harmless. However, `sparse` will complain if you define `gp` with `__rcu` and then access it without using either *rcu_access_pointer()* or rcu_dereference().

In short, RCU's publish-subscribe guarantee is provided by the combination of rcu_assign_pointer() and rcu_dereference(). This guarantee allows data elements to be safely added to RCU-protected linked data structures without disrupting RCU readers. This guarantee can be used in combination with the grace-period guarantee to also allow data elements to be removed from RCU-protected linked data structures, again without disrupting RCU readers.

This guarantee was only partially premeditated. DYNIX/ptx used an explicit memory barrier for publication, but had nothing resembling rcu_dereference() for subscription, nor did it have anything resembling the dependency-ordering barrier that was later subsumed into rcu_dereference() and later still into READ_ONCE(). The need for these operations made itself known quite suddenly at a late-1990s meeting with the DEC Alpha architects, back in the days when DEC was still a free-standing company. It took the Alpha architects a good hour to convince me that any sort of barrier would ever be needed, and it then took me a good *two* hours to convince them that their documentation did not make this point clear. More recent work with the C and C++ standards committees have provided much education on tricks and traps from the compiler. In short, compilers were much less tricky in the early 1990s, but in 2015, don't even think about omitting rcu_dereference()!

Memory-Barrier Guarantees

The previous section's simple linked-data-structure scenario clearly demonstrates the need for RCU's stringent memory-ordering guarantees on systems with more than one CPU:

1. Each CPU that has an RCU read-side critical section that begins before synchronize_rcu() starts is guaranteed to execute a full memory barrier between the time that the RCU read-side critical section ends and the time that synchronize_rcu() returns. Without this guarantee, a pre-existing RCU read-side critical section might hold a reference to the newly removed `struct foo` after the kfree() on line 14 of remove_gp_synchronous().
2. Each CPU that has an RCU read-side critical section that ends after synchronize_rcu() returns is guaranteed to execute a full memory barrier between the time that synchronize_rcu() begins and the time that the RCU read-side critical section begins. Without this guarantee, a later RCU read-side critical section running after the kfree() on line 14 of remove_gp_synchronous() might later run do_something_gp() and find the newly deleted `struct foo`.

3. If the task invoking synchronize_rcu() remains on a given CPU, then that CPU is guaranteed to execute a full memory barrier sometime during the execution of synchronize_rcu(). This guarantee ensures that the kfree() on line 14 of remove_gp_synchronous() really does execute after the removal on line 11.
4. If the task invoking synchronize_rcu() migrates among a group of CPUs during that invocation, then each of the CPUs in that group is guaranteed to execute a full memory barrier sometime during the execution of synchronize_rcu(). This guarantee also ensures that the kfree() on line 14 of remove_gp_synchronous() really does execute after the removal on line 11, but also in the case where the thread executing the synchronize_rcu() migrates in the meantime.

Quick Quiz:

Given that multiple CPUs can start RCU read-side critical sections at any time without any ordering whatsoever, how can RCU possibly tell whether or not a given RCU read-side critical section starts before a given instance of synchronize_rcu()?

Answer:

If RCU cannot tell whether or not a given RCU read-side critical section starts before a given instance of synchronize_rcu(), then it must assume that the RCU read-side critical section started first. In other words, a given instance of synchronize_rcu() can avoid waiting on a given RCU read-side critical section only if it can prove that synchronize_rcu() started first. A related question is "When rcu_read_lock() doesn't generate any code, why does it matter how it relates to a grace period?" The answer is that it is not the relationship of rcu_read_lock() itself that is important, but rather the relationship of the code within the enclosed RCU read-side critical section to the code preceding and following the grace period. If we take this viewpoint, then a given RCU read-side critical section begins before a given grace period when some access preceding the grace period observes the effect of some access within the critical section, in which case none of the accesses within the critical section may observe the effects of any access following the grace period.

As of late 2016, mathematical models of RCU take this viewpoint, for example, see slides 62 and 63 of the 2016 LinuxCon EU presentation.

Quick Quiz:
The first and second guarantees require unbelievably strict ordering! Are all these memory barriers *really* required?

Answer:
Yes, they really are required. To see why the first guarantee is required, consider the following sequence of events:

1. CPU 1: rcu_read_lock()
2. CPU 1: `q = rcu_dereference(gp); /* Very likely to return p. */`
3. CPU 0: `list_del_rcu(p);`
4. CPU 0: synchronize_rcu() starts.
5. CPU 1: `do_something_with(q->a); /* No smp_mb(), so might happen after kfree(). */`
6. CPU 1: rcu_read_unlock()
7. CPU 0: synchronize_rcu() returns.
8. CPU 0: `kfree(p);`

Therefore, there absolutely must be a full memory barrier between the end of the RCU read-side critical section and the end of the grace period.

The sequence of events demonstrating the necessity of the second rule is roughly similar:

1. CPU 0: `list_del_rcu(p);`
2. CPU 0: synchronize_rcu() starts.
3. CPU 1: rcu_read_lock()
4. CPU 1: `q = rcu_dereference(gp); /* Might return p if no memory barrier. */`
5. CPU 0: synchronize_rcu() returns.
6. CPU 0: `kfree(p);`
7. CPU 1: `do_something_with(q->a); /* Boom!!! */`
8. CPU 1: rcu_read_unlock()

And similarly, without a memory barrier between the beginning of the grace period and the beginning of the RCU read-side critical section, CPU 1 might end up accessing the freelist.

The "as if" rule of course applies, so that any implementation that acts as if the appropriate memory barriers were in place is a correct implementation. That said, it is much easier to fool yourself into believing that you have adhered to the as-if rule than it is to actually adhere to it!

Quick Quiz:
You claim that rcu_read_lock() and rcu_read_unlock() generate absolutely no code in some kernel builds. This means that the compiler might arbitrarily rearrange consecutive RCU read-side critical sections. Given such rearrangement, if a given RCU read-side critical section is done, how can you be sure that all prior RCU read-side critical sections are done? Won't the compiler rearrangements make that impossible to determine?

Answer:
In cases where rcu_read_lock() and rcu_read_unlock() generate absolutely no code, RCU infers quiescent states only at special locations, for example, within the scheduler. Because calls to schedule() had better prevent calling-code accesses to shared variables from being rearranged across the call to schedule(), if RCU detects the end of a given RCU read-side critical section, it will necessarily detect the end of all prior RCU read-side critical sections, no matter how aggressively the compiler scrambles the code. Again, this all assumes that the compiler cannot scramble code across calls to the scheduler, out of interrupt handlers, into the idle loop, into user-mode code, and so on. But if your kernel build allows that sort of scrambling, you have broken far more than just RCU!

Note that these memory-barrier requirements do not replace the fundamental RCU requirement that a grace period wait for all pre-existing readers. On the contrary, the memory barriers called out in this section must operate in such a way as to *enforce* this fundamental requirement. Of course, different implementations enforce this requirement in different ways, but enforce it they must.

RCU Primitives Guaranteed to Execute Unconditionally

The common-case RCU primitives are unconditional. They are invoked, they do their job, and they return, with no possibility of error, and no need to retry. This is a key RCU design philosophy.

However, this philosophy is pragmatic rather than pigheaded. If someone comes up with a good justification for a particular conditional RCU primitive, it might well be implemented and added. After all, this guarantee was reverse-engineered, not premeditated. The unconditional nature of the RCU primitives was initially an accident of implementation, and later experience with synchronization primitives with conditional primitives caused me to elevate this accident to a guarantee. Therefore, the justification for adding a conditional primitive to RCU would need to be based on detailed and compelling use cases.

Guaranteed Read-to-Write Upgrade

As far as RCU is concerned, it is always possible to carry out an update within an RCU read-side critical section. For example, that RCU read-side critical section might search for a given data element, and then might acquire the update-side spinlock in order to update that element, all while remaining in that RCU read-side critical section. Of course, it is necessary to exit the RCU read-side critical section before invoking synchronize_rcu(), however, this inconvenience can be avoided through use of the call_rcu() and kfree_rcu() API members described later in this document.

Quick Quiz:
But how does the upgrade-to-write operation exclude other readers?
Answer:
It doesn't, just like normal RCU updates, which also do not exclude RCU readers.

This guarantee allows lookup code to be shared between read-side and update-side code, and was premeditated, appearing in the earliest DYNIX/ptx RCU documentation.

Fundamental Non-Requirements

RCU provides extremely lightweight readers, and its read-side guarantees, though quite useful, are correspondingly lightweight. It is therefore all too easy to assume that RCU is guaranteeing more than it really is. Of course, the list of things that RCU does not guarantee is infinitely long, however, the following sections list a few non-guarantees that have caused confusion. Except where otherwise noted, these non-guarantees were premeditated.

1. *Readers Impose Minimal Ordering*
2. *Readers Do Not Exclude Updaters*

3. *Updaters Only Wait For Old Readers*
4. *Grace Periods Don't Partition Read-Side Critical Sections*
5. *Read-Side Critical Sections Don't Partition Grace Periods*

Readers Impose Minimal Ordering

Reader-side markers such as rcu_read_lock() and rcu_read_unlock() provide absolutely no ordering guarantees except through their interaction with the grace-period APIs such as synchronize_rcu(). To see this, consider the following pair of threads:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(x, 1);
  rcu_read_unlock();
  rcu_read_lock();
  WRITE_ONCE(y, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  rcu_read_lock();
  r1 = READ_ONCE(y);
  rcu_read_unlock();
  rcu_read_lock();
  r2 = READ_ONCE(x);
  rcu_read_unlock();
}
```

After thread0() and thread1() execute concurrently, it is quite possible to have

```
(r1 == 1 && r2 == 0)
```

(that is, y appears to have been assigned before x), which would not be possible if rcu_read_lock() and rcu_read_unlock() had much in the way of ordering properties. But they do not, so the CPU is within its rights to do significant reordering. This is by design: Any significant ordering constraints would slow down these fast-path APIs.

Quick Quiz:
Can't the compiler also reorder this code?
Answer:
No, the volatile casts in READ_ONCE() and WRITE_ONCE() prevent the compiler from reordering in this particular case.

Readers Do Not Exclude Updaters

Neither rcu_read_lock() nor rcu_read_unlock() exclude updates. All they do is to prevent grace periods from ending. The following example illustrates this:

```
void thread0(void)
{
  rcu_read_lock();
  r1 = READ_ONCE(y);
  if (r1) {
    do_something_with_nonzero_x();
    r2 = READ_ONCE(x);
    WARN_ON(!r2); /* BUG!!! */
  }
  rcu_read_unlock();
}

void thread1(void)
{
  spin_lock(&my_lock);
  WRITE_ONCE(x, 1);
  WRITE_ONCE(y, 1);
  spin_unlock(&my_lock);
}
```

If the thread0() function's rcu_read_lock() excluded the thread1() function's update, the WARN_ON() could never fire. But the fact is that rcu_read_lock() does not exclude much of anything aside from subsequent grace periods, of which thread1() has none, so the WARN_ON() can and does fire.

Updaters Only Wait For Old Readers

It might be tempting to assume that after synchronize_rcu() completes, there are no readers executing. This temptation must be avoided because new readers can start immediately after synchronize_rcu() starts, and synchronize_rcu() is under no obligation to wait for these new readers.

Quick Quiz:

Suppose that synchronize_rcu() did wait until *all* readers had completed instead of waiting only on pre-existing readers. For how long would the updater be able to rely on there being no readers?

Answer:

For no time at all. Even if synchronize_rcu() were to wait until all readers had completed, a new reader might start immediately after synchronize_rcu() completed. Therefore, the code following synchronize_rcu() can *never* rely on there being no readers.

Grace Periods Don't Partition Read-Side Critical Sections

It is tempting to assume that if any part of one RCU read-side critical section precedes a given grace period, and if any part of another RCU read-side critical section follows that same grace period, then all of the first RCU read-side critical section must precede all of the second. However, this just isn't the case: A single grace period does not partition the set of RCU read-side critical sections. An example of this situation can be illustrated as follows, where `x`, `y`, and `z` are initially all zero:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  rcu_read_lock();
  r2 = READ_ONCE(b);
  r3 = READ_ONCE(c);
  rcu_read_unlock();
}
```

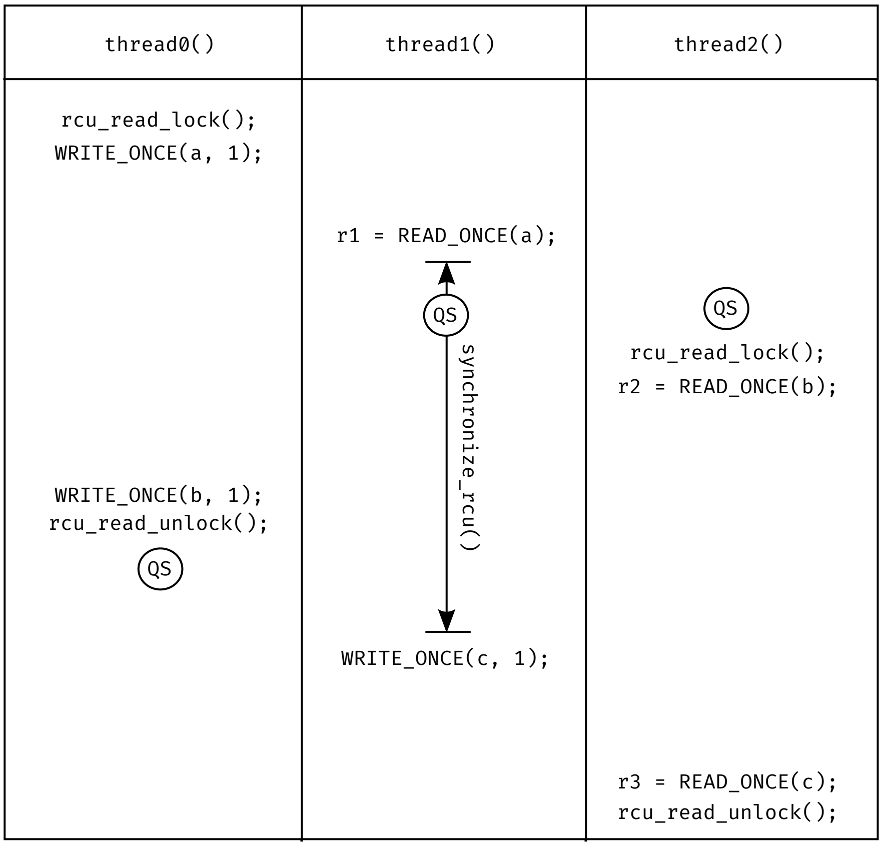
It turns out that the outcome:

```
(r1 == 1 && r2 == 0 && r3 == 1)
```

is entirely possible. The following figure show how this can happen, with each circled `QS` indicating the point at which RCU recorded a *quiescent state* for each thread, that is, a state in which RCU knows that the thread cannot be in the midst of an RCU read-side critical section that started before the current grace period:

If it is necessary to partition RCU read-side critical sections in this manner, it is necessary to use two grace periods, where the first grace period is known to end before the second grace period starts:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}
```




```
void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  r2 = READ_ONCE(c);
  synchronize_rcu();
  WRITE_ONCE(d, 1);
}

void thread3(void)
{
  rcu_read_lock();
  r3 = READ_ONCE(b);
  r4 = READ_ONCE(d);
  rcu_read_unlock();
}
```

Here, if `(r1 == 1)`, then thread0()'s write to `b` must happen before the end of thread1()'s grace period. If in addition `(r4 == 1)`, then thread3()'s read from `b` must happen after the beginning of thread2()'s grace period. If it is also the case that (r2 == 1), then the end of thread1()'s grace period must precede the beginning of thread2()'s grace period. This mean that the two RCU read-side critical sections cannot overlap, guaranteeing that (r3 == 1). As a result, the outcome:

```
(r1 == 1 && r2 == 1 && r3 == 0 && r4 == 1)
```

cannot happen.

This non-requirement was also non-premeditated, but became apparent when studying RCU's interaction with memory ordering.

Read-Side Critical Sections Don't Partition Grace Periods

It is also tempting to assume that if an RCU read-side critical section happens between a pair of grace periods, then those grace periods cannot overlap. However, this temptation leads nowhere good, as can be illustrated by the following, with all variables initially zero:

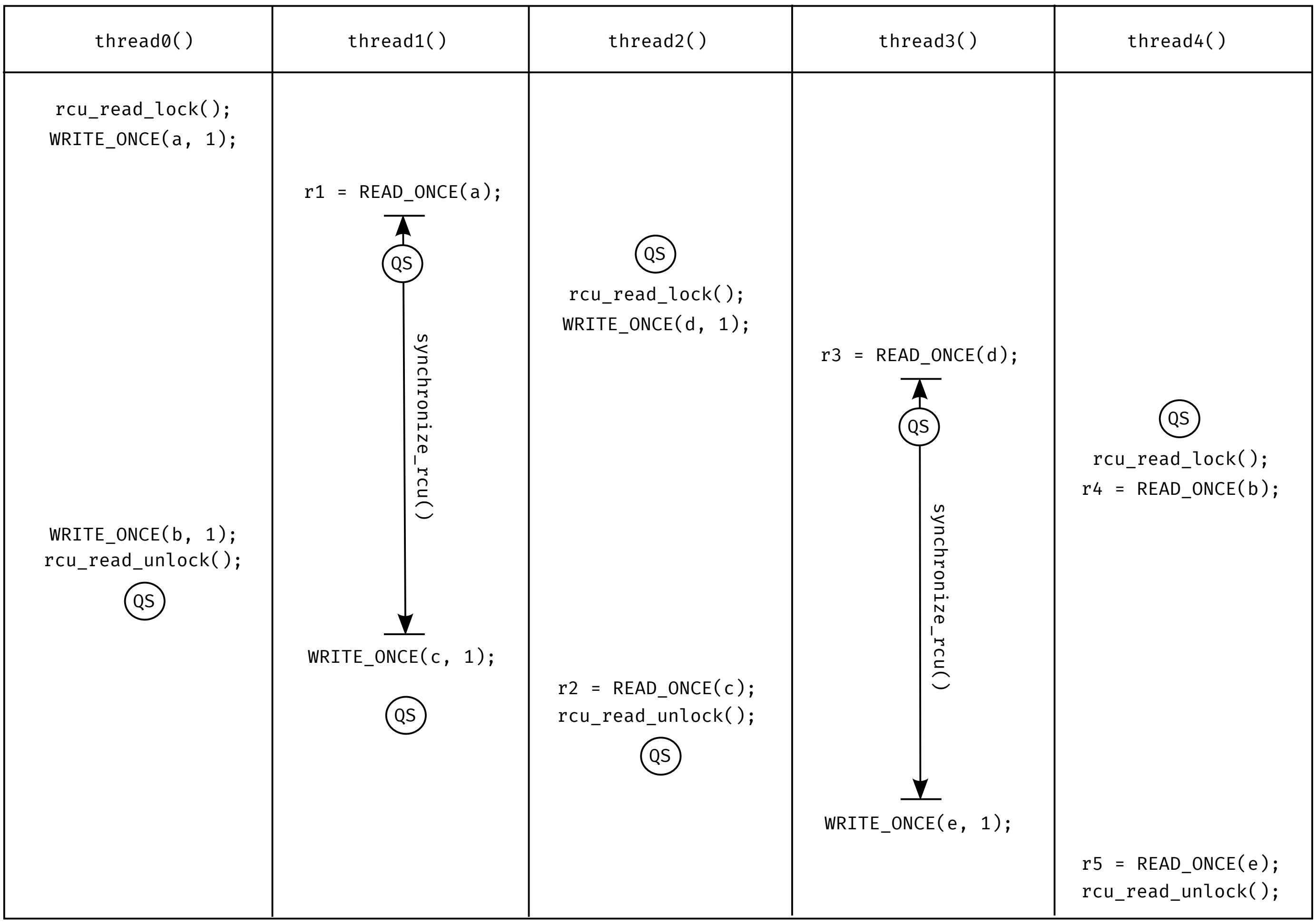
```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  rcu_read_lock();
  WRITE_ONCE(d, 1);
  r2 = READ_ONCE(c);
  rcu_read_unlock();
}

void thread3(void)
{
  r3 = READ_ONCE(d);
  synchronize_rcu();
  WRITE_ONCE(e, 1);
}

void thread4(void)
{
  rcu_read_lock();
```

```
  r4 = READ_ONCE(b);
  r5 = READ_ONCE(e);
  rcu_read_unlock();
}
```

In this case, the outcome:

```
(r1 == 1 && r2 == 1 && r3 == 1 && r4 == 0 && r5 == 1)
```

is entirely possible, as illustrated below:



Again, an RCU read-side critical section can overlap almost all of a given grace period, just so long as it does not overlap the entire grace period. As a result, an RCU read-side critical section cannot partition a pair of RCU grace periods.

Quick Quiz:

How long a sequence of grace periods, each separated by an RCU read-side critical section, would be required to partition the RCU read-side critical sections at the beginning and end of the chain?

Answer:

In theory, an infinite number. In practice, an unknown number that is sensitive to both implementation details and timing considerations. Therefore, even in practice, RCU users must abide by the theoretical rather than the practical answer.

Parallelism Facts of Life

These parallelism facts of life are by no means specific to RCU, but the RCU implementation must abide by them. They therefore bear repeating:

1. Any CPU or task may be delayed at any time, and any attempts to avoid these delays by disabling preemption, interrupts, or whatever are completely futile. This is most obvious in preemptible user-level environments and in virtualized environments (where a given guest OS's VCPUs can be preempted at any time by the underlying hypervisor), but can also happen in bare-metal environments due to ECC errors, NMIs, and other hardware events. Although a delay of more than about 20 seconds can result in splats, the RCU implementation is obligated to use algorithms that can tolerate extremely long delays, but where "extremely long" is not long enough to allow wrap-around when incrementing a 64-bit counter.
2. Both the compiler and the CPU can reorder memory accesses. Where it matters, RCU must use compiler directives and memory-barrier instructions to preserve ordering.
3. Conflicting writes to memory locations in any given cache line will result in expensive cache misses. Greater numbers of concurrent writes and more-frequent concurrent writes will result in more dramatic slowdowns. RCU is therefore obligated to use algorithms that have sufficient locality to avoid significant performance and scalability problems.
4. As a rough rule of thumb, only one CPU's worth of processing may be carried out under the protection of any given exclusive lock. RCU must therefore use scalable locking designs.
5. Counters are finite, especially on 32-bit systems. RCU's use of counters must therefore tolerate counter wrap, or be designed such that counter wrap would take way more time than a single system is likely to run. An uptime of ten years is quite possible, a runtime of a century much less so. As an example of the latter, RCU's dyntick-idle nesting counter allows 54 bits for interrupt nesting level (this counter is 64 bits even on a 32-bit system). Overflowing this counter requires 2^{54} half-interrupts on a given CPU without that CPU ever going idle. If a half-interrupt happened every microsecond, it would take 570 years of runtime to overflow this counter, which is currently believed to be an acceptably long time.
6. Linux systems can have thousands of CPUs running a single Linux kernel in a single shared-memory environment. RCU must therefore pay close attention to high-end scalability.

This last parallelism fact of life means that RCU must pay special attention to the preceding facts of life. The idea that Linux might scale to systems with thousands of CPUs would have been met with some skepticism in the 1990s, but these requirements would have otherwise have been unsurprising, even in the early 1990s.

Quality-of-Implementation Requirements

These sections list quality-of-implementation requirements. Although an RCU implementation that ignores these requirements could still be used, it would likely be subject to limitations that would make it inappropriate for industrial-strength production use. Classes of quality-of-implementation requirements are as follows:

1. *Specialization*
2. *Performance and Scalability*
3. *Forward Progress*

4. *Composability*
5. *Corner Cases*

These classes is covered in the following sections.

Specialization

RCU is and always has been intended primarily for read-mostly situations, which means that RCU's read-side primitives are optimized, often at the expense of its update-side primitives. Experience thus far is captured by the following list of situations:

1. Read-mostly data, where stale and inconsistent data is not a problem: RCU works great!
2. Read-mostly data, where data must be consistent: RCU works well.
3. Read-write data, where data must be consistent: RCU *might* work OK. Or not.
4. Write-mostly data, where data must be consistent: RCU is very unlikely to be the right tool for the job, with the following exceptions, where RCU can provide:
   a. Existence guarantees for update-friendly mechanisms.
   b. Wait-free read-side primitives for real-time use.

This focus on read-mostly situations means that RCU must interoperate with other synchronization primitives. For example, the add_gp() and remove_gp_synchronous() examples discussed earlier use RCU to protect readers and locking to coordinate updaters. However, the need extends much farther, requiring that a variety of synchronization primitives be legal within RCU read-side critical sections, including spinlocks, sequence locks, atomic operations, reference counters, and memory barriers.

Quick Quiz:
What about sleeping locks?
Answer:
These are forbidden within Linux-kernel RCU read-side critical sections because it is not legal to place a quiescent state (in this case, voluntary context switch) within an RCU read-side critical section. However, sleeping locks may be used within userspace RCU read-side critical sections, and also within Linux-kernel sleepable RCU *(SRCU)* read-side critical sections. In addition, the -rt patchset turns spinlocks into a sleeping locks so that the corresponding critical sections can be preempted, which also means that these sleeplockified spinlocks (but not other sleeping locks!) may be acquire within -rt-Linux-kernel RCU read-side critical sections. Note that it *is* legal for a normal RCU read-side critical section to conditionally acquire a sleeping locks (as in mutex_trylock()), but only as long as it does not loop indefinitely attempting to conditionally acquire that sleeping locks. The key point is that things like mutex_trylock() either return with the mutex held, or return an error indication if the mutex was not immediately available. Either way, mutex_trylock() returns immediately without sleeping.

It often comes as a surprise that many algorithms do not require a consistent view of data, but many can function in that mode, with network routing being the poster child. Internet routing algorithms take significant time to propagate updates, so that by the time an update arrives at a given system, that system has been sending network traffic the wrong way for a considerable length of time. Having a few threads continue to send traffic the wrong way for a few more milliseconds is clearly not a problem: In the worst case, TCP retransmissions will eventually get the data where it needs to go. In general, when tracking the state of the universe outside of the computer, some level of inconsistency must be tolerated due to speed-of-light delays if nothing else.

Furthermore, uncertainty about external state is inherent in many cases. For example, a pair of veterinarians might use heartbeat to determine whether or not a given cat was alive. But how long should they wait after the last heartbeat to decide that the cat is in fact dead? Waiting less than 400 milliseconds makes no sense because this would mean that a relaxed cat would be considered to cycle between death and life more than 100 times per minute. Moreover, just as with human beings, a cat's heart might stop for some period of time, so the exact wait period is a judgment call. One of our pair of veterinarians might wait 30 seconds before pronouncing the cat dead, while the other might insist on waiting a full minute. The two veterinarians would then disagree on the state of the cat during the final 30 seconds of the minute following the last heartbeat.

Interestingly enough, this same situation applies to hardware. When push comes to shove, how do we tell whether or not some external server has failed? We send messages to it periodically, and declare it failed if we don't receive a response within a given period of time. Policy decisions can usually tolerate short periods of inconsistency. The policy was decided some time ago, and is only now being put into effect, so a few milliseconds of delay is normally inconsequential.

However, there are algorithms that absolutely must see consistent data. For example, the translation between a user-level SystemV semaphore ID to the corresponding in-kernel data structure is protected by RCU, but it is absolutely forbidden to update a semaphore that has just been removed. In the Linux kernel, this need for consistency is accommodated by acquiring spinlocks located in the in-kernel data structure from within the RCU read-side critical section, and this is indicated by the green box in the figure above. Many other techniques may be used, and are in fact used within the Linux kernel.

In short, RCU is not required to maintain consistency, and other mechanisms may be used in concert with RCU when consistency is required. RCU's specialization allows it to do its job extremely well, and its ability to interoperate with other synchronization mechanisms allows the right mix of synchronization tools to be used for a given job.

Performance and Scalability

Energy efficiency is a critical component of performance today, and Linux-kernel RCU implementations must therefore avoid unnecessarily awakening idle CPUs. I cannot claim that this requirement was premeditated. In fact, I learned of it during a telephone conversation in which I was given "frank and open" feedback on the importance of energy efficiency in battery-powered systems and on specific energy-efficiency shortcomings of the Linux-kernel RCU implementation. In my experience, the battery-powered embedded community will consider any unnecessary wakeups to be extremely unfriendly acts. So much so that mere Linux-kernel-mailing-list posts are insufficient to vent their ire.

Memory consumption is not particularly important for in most situations, and has become decreasingly so as memory sizes have expanded and memory costs have plummeted. However, as I learned from Matt Mackall's bloatwatch efforts, memory footprint is critically important on single-CPU systems with non-preemptible (`CONFIG_PREEMPTION=n`) kernels, and thus tiny RCU was born. Josh Triplett has since taken over the small-memory banner with his Linux kernel tinification project, which resulted in *SRCU* becoming optional for those kernels not needing it.

The remaining performance requirements are, for the most part, unsurprising. For example, in keeping with RCU's read-side specialization, rcu_dereference() should have negligible overhead (for example, suppression of a few minor compiler optimizations). Similarly, in non-preemptible environments, rcu_read_lock() and rcu_read_unlock() should have exactly zero overhead.

In preemptible environments, in the case where the RCU read-side critical section was not preempted (as will be the case for the highest-priority real-time process), rcu_read_lock() and rcu_read_unlock() should have minimal overhead. In particular, they should not contain atomic read-modify-write operations, memory-barrier instructions, preemption disabling, interrupt disabling, or backwards branches. However, in the case where the RCU read-side critical section was preempted, rcu_read_unlock() may acquire spinlocks and disable interrupts. This is why it is better to nest an RCU read-side critical section within a preempt-disable region than vice versa, at least in cases where that critical section is short enough to avoid unduly degrading real-time latencies.

The synchronize_rcu() grace-period-wait primitive is optimized for throughput. It may therefore incur several milliseconds of latency in addition to the duration of the longest RCU read-side critical section. On the other hand, multiple concurrent invocations of synchronize_rcu() are required to use batching optimizations so that they can be satisfied by a single underlying grace-period-wait operation. For example, in the Linux kernel, it is not unusual for a single grace-period-wait operation to serve more than 1,000 separate invocations of synchronize_rcu(), thus amortizing the per-invocation overhead down to nearly zero. However, the grace-period optimization is also required to avoid measurable degradation of real-time scheduling and interrupt latencies.

In some cases, the multi-millisecond synchronize_rcu() latencies are unacceptable. In these cases, *synchronize_rcu_expedited()* may be used instead, reducing the grace-period latency down to a few tens of microseconds on small systems, at least in cases where the RCU read-side critical sections are short. There are currently no special latency requirements for *synchronize_rcu_expedited()* on large systems, but, consistent with the empirical nature of the RCU specification, that is subject to change. However, there most definitely are scalability requirements: A storm of *synchronize_rcu_expedited()* invocations on 4096 CPUs should at least make reasonable forward progress. In return for its shorter latencies, *synchronize_rcu_expedited()* is permitted to impose modest degradation of real-time latency on non-idle online CPUs. Here, "modest" means roughly the same latency degradation as a scheduling-clock interrupt.

There are a number of situations where even *synchronize_rcu_expedited()*'s reduced grace-period latency is unacceptable. In these situations, the asynchronous call_rcu() can be used in place of synchronize_rcu() as follows:

```
struct foo {
  int a;
  int b;
  struct rcu_head rh;
};

static void remove_gp_cb(struct rcu_head *rhp)
{
  struct foo *p = container_of(rhp, struct foo, rh);

  kfree(p);
}
```

```
13
14 bool remove_gp_asynchronous(void)
15 {
16   struct foo *p;
17
18   spin_lock(&gp_lock);
19   p = rcu_access_pointer(gp);
20   if (!p) {
21     spin_unlock(&gp_lock);
22     return false;
23   }
24   rcu_assign_pointer(gp, NULL);
25   call_rcu(&p->rh, remove_gp_cb);
26   spin_unlock(&gp_lock);
27   return true;
28 }
```

A definition of `struct foo` is finally needed, and appears on lines 1-5. The function remove_gp_cb() is passed to call_rcu() on line 25, and will be invoked after the end of a subsequent grace period. This gets the same effect as remove_gp_synchronous(), but without forcing the updater to wait for a grace period to elapse. The call_rcu() function may be used in a number of situations where neither synchronize_rcu() nor *synchronize_rcu_expedited()* would be legal, including within preempt-disable code, local_bh_disable() code, interrupt-disable code, and interrupt handlers. However, even call_rcu() is illegal within NMI handlers and from idle and offline CPUs. The callback function (remove_gp_cb() in this case) will be executed within softirq (software interrupt) environment within the Linux kernel, either within a real softirq handler or under the protection of local_bh_disable(). In both the Linux kernel and in userspace, it is bad practice to write an RCU callback function that takes too long. Long-running operations should be relegated to separate threads or (in the Linux kernel) workqueues.

Quick Quiz:

Why does line 19 use *rcu_access_pointer()*? After all, call_rcu() on line 25 stores into the structure, which would interact badly with concurrent insertions. Doesn't this mean that rcu_dereference() is required?

Answer:

Presumably the `->gp_lock` acquired on line 18 excludes any changes, including any insertions that rcu_dereference() would protect against. Therefore, any insertions will be delayed until after `->gp_lock` is released on line 25, which in turn means that *rcu_access_pointer()* suffices.

However, all that remove_gp_cb() is doing is invoking kfree() on the data element. This is a common idiom, and is supported by kfree_rcu(), which allows "fire and forget" operation as shown below:

```
struct foo {
  int a;
  int b;
  struct rcu_head rh;
};

```

```
bool remove_gp_faf(void)
{
  struct foo *p;

  spin_lock(&gp_lock);
  p = rcu_dereference(gp);
  if (!p) {
    spin_unlock(&gp_lock);
    return false;
  }
  rcu_assign_pointer(gp, NULL);
  kfree_rcu(p, rh);
  spin_unlock(&gp_lock);
  return true;
}
```

Note that remove_gp_faf() simply invokes kfree_rcu() and proceeds, without any need to pay any further attention to the subsequent grace period and kfree(). It is permissible to invoke kfree_rcu() from the same environments as for call_rcu(). Interestingly enough, DYNIX/ptx had the equivalents of call_rcu() and kfree_rcu(), but not synchronize_rcu(). This was due to the fact that RCU was not heavily used within DYNIX/ptx, so the very few places that needed something like synchronize_rcu() simply open-coded it.

Quick Quiz:

Earlier it was claimed that call_rcu() and kfree_rcu() allowed updaters to avoid being blocked by readers. But how can that be correct, given that the invocation of the callback and the freeing of the memory (respectively) must still wait for a grace period to elapse?

Answer:

We could define things this way, but keep in mind that this sort of definition would say that updates in garbage-collected languages cannot complete until the next time the garbage collector runs, which does not seem at all reasonable. The key point is that in most cases, an updater using either call_rcu() or kfree_rcu() can proceed to the next update as soon as it has invoked call_rcu() or kfree_rcu(), without having to wait for a subsequent grace period.

But what if the updater must wait for the completion of code to be executed after the end of the grace period, but has other tasks that can be carried out in the meantime? The polling-style *get_state_synchronize_rcu()* and *cond_synchronize_rcu()* functions may be used for this purpose, as shown below:

```
bool remove_gp_poll(void)
{
  struct foo *p;
  unsigned long s;

  spin_lock(&gp_lock);
  p = rcu_access_pointer(gp);
  if (!p) {
    spin_unlock(&gp_lock);
    return false;
  }
```

```
12   rcu_assign_pointer(gp, NULL);
13   spin_unlock(&gp_lock);
14   s = get_state_synchronize_rcu();
15   do_something_while_waiting();
16   cond_synchronize_rcu(s);
17   kfree(p);
18   return true;
19 }
```

On line 14, *get_state_synchronize_rcu()* obtains a "cookie" from RCU, then line 15 carries out other tasks, and finally, line 16 returns immediately if a grace period has elapsed in the meantime, but otherwise waits as required. The need for `get_state_synchronize_rcu` and *cond_synchronize_rcu()* has appeared quite recently, so it is too early to tell whether they will stand the test of time.

RCU thus provides a range of tools to allow updaters to strike the required tradeoff between latency, flexibility and CPU overhead.

Forward Progress

In theory, delaying grace-period completion and callback invocation is harmless. In practice, not only are memory sizes finite but also callbacks sometimes do wakeups, and sufficiently deferred wakeups can be difficult to distinguish from system hangs. Therefore, RCU must provide a number of mechanisms to promote forward progress.

These mechanisms are not foolproof, nor can they be. For one simple example, an infinite loop in an RCU read-side critical section must by definition prevent later grace periods from ever completing. For a more involved example, consider a 64-CPU system built with `CONFIG_RCU_NOCB_CPU=y` and booted with `rcu_nocbs=1-63`, where CPUs 1 through 63 spin in tight loops that invoke call_rcu(). Even if these tight loops also contain calls to cond_resched() (thus allowing grace periods to complete), CPU 0 simply will not be able to invoke callbacks as fast as the other 63 CPUs can register them, at least not until the system runs out of memory. In both of these examples, the Spiderman principle applies: With great power comes great responsibility. However, short of this level of abuse, RCU is required to ensure timely completion of grace periods and timely invocation of callbacks.

RCU takes the following steps to encourage timely completion of grace periods:

1. If a grace period fails to complete within 100 milliseconds, RCU causes future invocations of cond_resched() on the holdout CPUs to provide an RCU quiescent state. RCU also causes those CPUs' need_resched() invocations to return `true`, but only after the corresponding CPU's next scheduling-clock.
2. CPUs mentioned in the `nohz_full` kernel boot parameter can run indefinitely in the kernel without scheduling-clock interrupts, which defeats the above need_resched() strategem. RCU will therefore invoke resched_cpu() on any `nohz_full` CPUs still holding out after 109 milliseconds.
3. In kernels built with `CONFIG_RCU_BOOST=y`, if a given task that has been preempted within an RCU read-side critical section is holding out for more than 500 milliseconds, RCU will resort to priority boosting.

4. If a CPU is still holding out 10 seconds into the grace period, RCU will invoke resched_cpu() on it regardless of its `nohz_full` state.

The above values are defaults for systems running with `HZ=1000`. They will vary as the value of `HZ` varies, and can also be changed using the relevant Kconfig options and kernel boot parameters. RCU currently does not do much sanity checking of these parameters, so please use caution when changing them. Note that these forward-progress measures are provided only for RCU, not for *SRCU* or *Tasks RCU*.

RCU takes the following steps in call_rcu() to encourage timely invocation of callbacks when any given non-`rcu_nocbs` CPU has 10,000 callbacks, or has 10,000 more callbacks than it had the last time encouragement was provided:

1. Starts a grace period, if one is not already in progress.
2. Forces immediate checking for quiescent states, rather than waiting for three milliseconds to have elapsed since the beginning of the grace period.
3. Immediately tags the CPU's callbacks with their grace period completion numbers, rather than waiting for the `RCU_SOFTIRQ` handler to get around to it.
4. Lifts callback-execution batch limits, which speeds up callback invocation at the expense of degrading realtime response.

Again, these are default values when running at `HZ=1000`, and can be overridden. Again, these forward-progress measures are provided only for RCU, not for *SRCU* or *Tasks RCU*. Even for RCU, callback-invocation forward progress for `rcu_nocbs` CPUs is much less well-developed, in part because workloads benefiting from `rcu_nocbs` CPUs tend to invoke call_rcu() relatively infrequently. If workloads emerge that need both `rcu_nocbs` CPUs and high call_rcu() invocation rates, then additional forward-progress work will be required.

Composability

Composability has received much attention in recent years, perhaps in part due to the collision of multicore hardware with object-oriented techniques designed in single-threaded environments for single-threaded use. And in theory, RCU read-side critical sections may be composed, and in fact may be nested arbitrarily deeply. In practice, as with all real-world implementations of composable constructs, there are limitations.

Implementations of RCU for which rcu_read_lock() and rcu_read_unlock() generate no code, such as Linux-kernel RCU when `CONFIG_PREEMPTION=n`, can be nested arbitrarily deeply. After all, there is no overhead. Except that if all these instances of rcu_read_lock() and rcu_read_unlock() are visible to the compiler, compilation will eventually fail due to exhausting memory, mass storage, or user patience, whichever comes first. If the nesting is not visible to the compiler, as is the case with mutually recursive functions each in its own translation unit, stack overflow will result. If the nesting takes the form of loops, perhaps in the guise of tail recursion, either the control variable will overflow or (in the Linux kernel) you will get an RCU CPU stall warning. Nevertheless, this class of RCU implementations is one of the most composable constructs in existence.

RCU implementations that explicitly track nesting depth are limited by the nesting-depth counter. For example, the Linux kernel's preemptible RCU limits nesting to `INT_MAX`. This should suffice for almost all practical purposes. That said, a consecutive pair of RCU read-side critical sections between which there is an operation that waits for a grace period cannot be enclosed in another RCU read-side critical section. This is because it is not legal to wait for a grace period within an RCU read-side critical section: To do so would result either in deadlock or in RCU implicitly splitting the enclosing RCU read-side critical section, neither of which is conducive to a long-lived and prosperous kernel.

It is worth noting that RCU is not alone in limiting composability. For example, many transactional-memory implementations prohibit composing a pair of transactions separated by an irrevocable operation (for example, a network receive operation). For another example, lock-based critical sections can be composed surprisingly freely, but only if deadlock is avoided.

In short, although RCU read-side critical sections are highly composable, care is required in some situations, just as is the case for any other composable synchronization mechanism.

Corner Cases

A given RCU workload might have an endless and intense stream of RCU read-side critical sections, perhaps even so intense that there was never a point in time during which there was not at least one RCU read-side critical section in flight. RCU cannot allow this situation to block grace periods: As long as all the RCU read-side critical sections are finite, grace periods must also be finite.

That said, preemptible RCU implementations could potentially result in RCU read-side critical sections being preempted for long durations, which has the effect of creating a long-duration RCU read-side critical section. This situation can arise only in heavily loaded systems, but systems using real-time priorities are of course more vulnerable. Therefore, RCU priority boosting is provided to help deal with this case. That said, the exact requirements on RCU priority boosting will likely evolve as more experience accumulates.

Other workloads might have very high update rates. Although one can argue that such workloads should instead use something other than RCU, the fact remains that RCU must handle such workloads gracefully. This requirement is another factor driving batching of grace periods, but it is also the driving force behind the checks for large numbers of queued RCU callbacks in the call_rcu() code path. Finally, high update rates should not delay RCU read-side critical sections, although some small read-side delays can occur when using *synchronize_rcu_expedited()*, courtesy of this function's use of smp_call_function_single().

Although all three of these corner cases were understood in the early 1990s, a simple user-level test consisting of `close(open(path))` in a tight loop in the early 2000s suddenly provided a much deeper appreciation of the high-update-rate corner case. This test also motivated addition of some RCU code to react to high update rates, for example, if a given CPU finds itself with more than 10,000 RCU callbacks queued, it will cause RCU to take evasive action by more aggressively starting grace periods and more aggressively forcing completion of grace-period processing. This evasive action causes the grace period to complete more quickly, but at the cost of restricting RCU's batching optimizations, thus increasing the CPU overhead incurred by that grace period.

Software-Engineering Requirements

Between Murphy's Law and "To err is human", it is necessary to guard against mishaps and misuse:

1. It is all too easy to forget to use rcu_read_lock() everywhere that it is needed, so kernels built with `CONFIG_PROVE_RCU=y` will splat if rcu_dereference() is used outside of an RCU read-side critical section. Update-side code can use *rcu_dereference_protected()*, which takes a lockdep expression to indicate what is providing the protection. If the indicated protection is not provided, a lockdep splat is emitted. Code shared between readers and updaters can use *rcu_dereference_check()*, which also takes a lockdep expression, and emits a lockdep splat if neither rcu_read_lock() nor the indicated protection is in place. In addition, rcu_dereference_raw() is used in those (hopefully rare) cases where the required protection cannot be easily described. Finally, *rcu_read_lock_held()* is provided to allow a function to verify that it has been invoked within an RCU read-side critical section. I was made aware of this set of requirements shortly after Thomas Gleixner audited a number of RCU uses.

2. A given function might wish to check for RCU-related preconditions upon entry, before using any other RCU API. The rcu_lockdep_assert() does this job, asserting the expression in kernels having lockdep enabled and doing nothing otherwise.

3. It is also easy to forget to use rcu_assign_pointer() and rcu_dereference(), perhaps (incorrectly) substituting a simple assignment. To catch this sort of error, a given RCU-protected pointer may be tagged with `__rcu`, after which sparse will complain about simple-assignment accesses to that pointer. Arnd Bergmann made me aware of this requirement, and also supplied the needed patch series.

4. Kernels built with `CONFIG_DEBUG_OBJECTS_RCU_HEAD=y` will splat if a data element is passed to call_rcu() twice in a row, without a grace period in between. (This error is similar to a double free.) The corresponding `rcu_head` structures that are dynamically allocated are automatically tracked, but `rcu_head` structures allocated on the stack must be initialized with *init_rcu_head_on_stack()* and cleaned up with *destroy_rcu_head_on_stack()*. Similarly, statically allocated non-stack `rcu_head` structures must be initialized with init_rcu_head() and cleaned up with destroy_rcu_head(). Mathieu Desnoyers made me aware of this requirement, and also supplied the needed patch.

5. An infinite loop in an RCU read-side critical section will eventually trigger an RCU CPU stall warning splat, with the duration of "eventually" being controlled by the `RCU_CPU_STALL_TIMEOUT` `Kconfig` option, or, alternatively, by the `rcupdate.rcu_cpu_stall_timeout` boot/sysfs parameter. However, RCU is not obligated to produce this splat unless there is a grace period waiting on that particular RCU read-side critical section.

   Some extreme workloads might intentionally delay RCU grace periods, and systems running those workloads can be booted with `rcupdate.rcu_cpu_stall_suppress` to suppress the splats. This kernel parameter may also be set via `sysfs`. Furthermore, RCU CPU stall warnings are counter-productive during sysrq dumps and during panics. RCU therefore supplies the rcu_sysrq_start() and rcu_sysrq_end() API members to be called before and after long sysrq dumps. RCU also supplies the rcu_panic() notifier that is automatically invoked at the beginning of a panic to suppress further RCU CPU stall warnings.

   This requirement made itself known in the early 1990s, pretty much the first time that it was necessary to debug a CPU stall. That said, the initial implementation in DYNIX/ptx was quite generic in comparison with that of Linux.

6. Although it would be very good to detect pointers leaking out of RCU read-side critical sections, there is currently no good way of doing this. One complication is the need to distinguish between pointers leaking and pointers that have been handed off from RCU to some other synchronization mechanism, for example, reference counting.
7. In kernels built with `CONFIG_RCU_TRACE=y`, RCU-related information is provided via event tracing.
8. Open-coded use of rcu_assign_pointer() and rcu_dereference() to create typical linked data structures can be surprisingly error-prone. Therefore, RCU-protected linked lists and, more recently, RCU-protected hash tables are available. Many other special-purpose RCU-protected data structures are available in the Linux kernel and the userspace RCU library.
9. Some linked structures are created at compile time, but still require `__rcu` checking. The *RCU_POINTER_INITIALIZER()* macro serves this purpose.
10. It is not necessary to use rcu_assign_pointer() when creating linked structures that are to be published via a single external pointer. The *RCU_INIT_POINTER()* macro is provided for this task.

This not a hard-and-fast list: RCU's diagnostic capabilities will continue to be guided by the number and type of usage bugs found in real-world RCU usage.

Linux Kernel Complications

The Linux kernel provides an interesting environment for all kinds of software, including RCU. Some of the relevant points of interest are as follows:

1. *Configuration*
2. *Firmware Interface*
3. *Early Boot*
4. *Interrupts and NMIs*
5. *Loadable Modules*
6. *Hotplug CPU*
7. *Scheduler and RCU*
8. *Tracing and RCU*
9. *Accesses to User Memory and RCU*
10. *Energy Efficiency*
11. *Scheduling-Clock Interrupts and RCU*
12. *Memory Efficiency*
13. *Performance, Scalability, Response Time, and Reliability*

This list is probably incomplete, but it does give a feel for the most notable Linux-kernel complications. Each of the following sections covers one of the above topics.

Configuration

RCU's goal is automatic configuration, so that almost nobody needs to worry about RCU's Kconfig options. And for almost all users, RCU does in fact work well "out of the box."

However, there are specialized use cases that are handled by kernel boot parameters and Kconfig options. Unfortunately, the Kconfig system will explicitly ask users about new Kconfig options, which requires almost all of them be hidden behind a CONFIG_RCU_EXPERT Kconfig option.

This all should be quite obvious, but the fact remains that Linus Torvalds recently had to remind me of this requirement.

Firmware Interface

In many cases, kernel obtains information about the system from the firmware, and sometimes things are lost in translation. Or the translation is accurate, but the original message is bogus.

For example, some systems' firmware overreports the number of CPUs, sometimes by a large factor. If RCU naively believed the firmware, as it used to do, it would create too many per-CPU kthreads. Although the resulting system will still run correctly, the extra kthreads needlessly consume memory and can cause confusion when they show up in ps listings.

RCU must therefore wait for a given CPU to actually come online before it can allow itself to believe that the CPU actually exists. The resulting "ghost CPUs" (which are never going to come online) cause a number of interesting complications.

Early Boot

The Linux kernel's boot sequence is an interesting process, and RCU is used early, even before rcu_init() is invoked. In fact, a number of RCU's primitives can be used as soon as the initial task's task_struct is available and the boot CPU's per-CPU variables are set up. The read-side primitives (rcu_read_lock(), rcu_read_unlock(), rcu_dereference(), and *rcu_access_pointer()*) will operate normally very early on, as will rcu_assign_pointer().

Although call_rcu() may be invoked at any time during boot, callbacks are not guaranteed to be invoked until after all of RCU's kthreads have been spawned, which occurs at early_initcall() time. This delay in callback invocation is due to the fact that RCU does not invoke callbacks until it is fully initialized, and this full initialization cannot occur until after the scheduler has initialized itself to the point where RCU can spawn and run its kthreads. In theory, it would be possible to invoke callbacks earlier, however, this is not a panacea because there would be severe restrictions on what operations those callbacks could invoke.

Perhaps surprisingly, synchronize_rcu() and *synchronize_rcu_expedited()*, will operate normally during very early boot, the reason being that there is only one CPU and preemption is disabled. This means that the call synchronize_rcu() (or friends) itself is a quiescent state and thus a grace period, so the early-boot implementation can be a no-op.

However, once the scheduler has spawned its first kthread, this early boot trick fails for synchronize_rcu() (as well as for *synchronize_rcu_expedited()*) in CONFIG_PREEMPTION=y kernels. The reason is that an RCU read-side critical section might be preempted, which means that a subsequent synchronize_rcu() really does have to wait for something, as opposed to simply returning immediately. Unfortunately, synchronize_rcu() can't do this until all of its kthreads are spawned, which doesn't happen until some time during early_initcalls() time. But this is no excuse: RCU is nevertheless required to correctly handle synchronous grace periods during this time period. Once all of its kthreads are up and running, RCU starts running normally.

Quick Quiz:
How can RCU possibly handle grace periods before all of its kthreads have been spawned???
Answer:
Very carefully! During the "dead zone" between the time that the scheduler spawns the first task and the time that all of RCU's kthreads have been spawned, all synchronous grace periods are handled by the expedited grace-period mechanism. At runtime, this expedited mechanism relies on workqueues, but during the dead zone the requesting task itself drives the desired expedited grace period. Because dead-zone execution takes place within task context, everything works. Once the dead zone ends, expedited grace periods go back to using workqueues, as is required to avoid problems that would otherwise occur when a user task received a POSIX signal while driving an expedited grace period. And yes, this does mean that it is unhelpful to send POSIX signals to random tasks between the time that the scheduler spawns its first kthread and the time that RCU's kthreads have all been spawned. If there ever turns out to be a good reason for sending POSIX signals during that time, appropriate adjustments will be made. (If it turns out that POSIX signals are sent during this time for no good reason, other adjustments will be made, appropriate or otherwise.)

I learned of these boot-time requirements as a result of a series of system hangs.

Interrupts and NMIs

The Linux kernel has interrupts, and RCU read-side critical sections are legal within interrupt handlers and within interrupt-disabled regions of code, as are invocations of call_rcu().

Some Linux-kernel architectures can enter an interrupt handler from non-idle process context, and then just never leave it, instead stealthily transitioning back to process context. This trick is sometimes used to invoke system calls from inside the kernel. These "half-interrupts" mean that RCU has to be very careful about how it counts interrupt nesting levels. I learned of this requirement the hard way during a rewrite of RCU's dyntick-idle code.

The Linux kernel has non-maskable interrupts (NMIs), and RCU read-side critical sections are legal within NMI handlers. Thankfully, RCU update-side primitives, including call_rcu(), are prohibited within NMI handlers.

The name notwithstanding, some Linux-kernel architectures can have nested NMIs, which RCU must handle correctly. Andy Lutomirski surprised me with this requirement; he also kindly surprised me with an algorithm that meets this requirement.

Furthermore, NMI handlers can be interrupted by what appear to RCU to be normal interrupts. One way that this can happen is for code that directly invokes ct_irq_enter() and ct_irq_exit() to be called from an NMI handler. This astonishing fact of life prompted the current code structure, which has ct_irq_enter() invoking ct_nmi_enter() and ct_irq_exit() invoking ct_nmi_exit(). And yes, I also learned of this requirement the hard way.

Loadable Modules

The Linux kernel has loadable modules, and these modules can also be unloaded. After a given module has been unloaded, any attempt to call one of its functions results in a segmentation fault. The module-unload functions must therefore cancel any delayed calls to loadable-module functions, for example, any outstanding mod_timer() must be dealt with via timer_shutdown_sync() or similar.

Unfortunately, there is no way to cancel an RCU callback; once you invoke call_rcu(), the callback function is eventually going to be invoked, unless the system goes down first. Because it is normally considered socially irresponsible to crash the system in response to a module unload request, we need some other way to deal with in-flight RCU callbacks.

RCU therefore provides *rcu_barrier()*, which waits until all in-flight RCU callbacks have been invoked. If a module uses call_rcu(), its exit function should therefore prevent any future invocation of call_rcu(), then invoke *rcu_barrier()*. In theory, the underlying module-unload code could invoke *rcu_barrier()* unconditionally, but in practice this would incur unacceptable latencies.

Nikita Danilov noted this requirement for an analogous filesystem-unmount situation, and Dipankar Sarma incorporated *rcu_barrier()* into RCU. The need for *rcu_barrier()* for module unloading became apparent later.

Important: The *rcu_barrier()* function is not, repeat, *not*, obligated to wait for a grace period. It is instead only required to wait for RCU callbacks that have already been posted. Therefore, if there are no RCU callbacks posted anywhere in the system, *rcu_barrier()* is within its rights to return immediately. Even if there are callbacks posted, *rcu_barrier()* does not necessarily need to wait for a grace period.

Quick Quiz:

Wait a minute! Each RCU callbacks must wait for a grace period to complete, and *rcu_barrier()* must wait for each pre-existing callback to be invoked. Doesn't *rcu_barrier()* therefore need to wait for a full grace period if there is even one callback posted anywhere in the system?

Answer:

Absolutely not!!! Yes, each RCU callbacks must wait for a grace period to complete, but it might well be partly (or even completely) finished waiting by the time *rcu_barrier()* is invoked. In that case, *rcu_barrier()* need only wait for the remaining portion of the grace period to elapse. So even if there are quite a few callbacks posted, *rcu_barrier()* might well return quite quickly.

So if you need to wait for a grace period as well as for all pre-existing callbacks, you will need to invoke both synchronize_rcu() and *rcu_barrier()*. If latency is a concern, you can always use workqueues to invoke them concurrently.

Hotplug CPU

The Linux kernel supports CPU hotplug, which means that CPUs can come and go. It is of course illegal to use any RCU API member from an offline CPU, with the exception of *SRCU* read-side critical sections. This requirement was present from day one in DYNIX/ptx, but on the other hand, the Linux kernel's CPU-hotplug implementation is "interesting."

The Linux-kernel CPU-hotplug implementation has notifiers that are used to allow the various kernel subsystems (including RCU) to respond appropriately to a given CPU-hotplug operation. Most RCU operations may be invoked from CPU-hotplug notifiers, including even synchronous grace-period operations such as (synchronize_rcu() and `synchronize_rcu_expedited()`). However, these synchronous operations do block and therefore cannot be invoked from notifiers that execute via stop_machine(), specifically those between the `CPUHP_AP_OFFLINE` and `CPUHP_AP_ONLINE` states.

In addition, all-callback-wait operations such as `rcu_barrier()` may not be invoked from any CPU-hotplug notifier. This restriction is due to the fact that there are phases of CPU-hotplug operations where the outgoing CPU's callbacks will not be invoked until after the CPU-hotplug operation ends, which could also result in deadlock. Furthermore, `rcu_barrier()` blocks CPU-hotplug operations during its execution, which results in another type of deadlock when invoked from a CPU-hotplug notifier.

Finally, RCU must avoid deadlocks due to interaction between hotplug, timers and grace period processing. It does so by maintaining its own set of books that duplicate the centrally maintained `cpu_online_mask`, and also by reporting quiescent states explicitly when a CPU goes offline. This explicit reporting of quiescent states avoids any need for the force-quiescent-state loop (FQS) to report quiescent states for offline CPUs. However, as a debugging measure, the FQS loop does splat if offline CPUs block an RCU grace period for too long.

An offline CPU's quiescent state will be reported either:

1. As the CPU goes offline using RCU's hotplug notifier (rcu_report_dead()).
2. When grace period initialization (rcu_gp_init()) detects a race either with CPU offlining or with a task unblocking on a leaf `rcu_node` structure whose CPUs are all offline.

The CPU-online path (rcu_cpu_starting()) should never need to report a quiescent state for an offline CPU. However, as a debugging measure, it does emit a warning if a quiescent state was not already reported for that CPU.

During the checking/modification of RCU's hotplug bookkeeping, the corresponding CPU's leaf node lock is held. This avoids race conditions between RCU's hotplug notifier hooks, the grace period initialization code, and the FQS loop, all of which refer to or modify this bookkeeping.

Scheduler and RCU

RCU makes use of kthreads, and it is necessary to avoid excessive CPU-time accumulation by these kthreads. This requirement was no surprise, but RCU's violation of it when running context-switch-heavy workloads when built with `CONFIG_NO_HZ_FULL=y` did come as a surprise [PDF]. RCU has made good progress towards meeting this requirement, even for context-switch-heavy `CONFIG_NO_HZ_FULL=y` workloads, but there is room for further improvement.

There is no longer any prohibition against holding any of scheduler's runqueue or priority-inheritance spinlocks across an rcu_read_unlock(), even if interrupts and preemption were enabled somewhere within the corresponding RCU read-side critical section. Therefore, it is now perfectly legal to execute rcu_read_lock() with preemption enabled, acquire one of the scheduler locks, and hold that lock across the matching rcu_read_unlock().

Similarly, the RCU flavor consolidation has removed the need for negative nesting. The fact that interrupt-disabled regions of code act as RCU read-side critical sections implicitly avoids earlier issues that used to result in destructive recursion via interrupt handler's use of RCU.

Tracing and RCU

It is possible to use tracing on RCU code, but tracing itself uses RCU. For this reason, rcu_dereference_raw_check() is provided for use by tracing, which avoids the destructive recursion that could otherwise ensue. This API is also used by virtualization in some architectures, where RCU readers execute in environments in which tracing cannot be used. The tracing folks both located the requirement and provided the needed fix, so this surprise requirement was relatively painless.

Accesses to User Memory and RCU

The kernel needs to access user-space memory, for example, to access data referenced by system-call parameters. The get_user() macro does this job.

However, user-space memory might well be paged out, which means that get_user() might well page-fault and thus block while waiting for the resulting I/O to complete. It would be a very bad thing for the compiler to reorder a get_user() invocation into an RCU read-side critical section.

For example, suppose that the source code looked like this:

```
rcu_read_lock();
p = rcu_dereference(gp);
v = p->value;
rcu_read_unlock();
get_user(user_v, user_p);
do_something_with(v, user_v);
```

The compiler must not be permitted to transform this source code into the following:

```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 get_user(user_v, user_p); // BUG: POSSIBLE PAGE FAULT!!!
4 v = p->value;
5 rcu_read_unlock();
6 do_something_with(v, user_v);
```

If the compiler did make this transformation in a `CONFIG_PREEMPTION=n` kernel build, and if get_user() did page fault, the result would be a quiescent state in the middle of an RCU read-side critical section. This misplaced quiescent state could result in line 4 being a use-after-free access, which could be bad for your kernel's actuarial statistics. Similar examples can be constructed with the call to get_user() preceding the rcu_read_lock().

Unfortunately, get_user() doesn't have any particular ordering properties, and in some architectures the underlying `asm` isn't even marked `volatile`. And even if it was marked `volatile`, the above access to p->value is not volatile, so the compiler would not have any reason to keep those two accesses in order.

Therefore, the Linux-kernel definitions of rcu_read_lock() and rcu_read_unlock() must act as compiler barriers, at least for outermost instances of rcu_read_lock() and rcu_read_unlock() within a nested set of RCU read-side critical sections.

Energy Efficiency

Interrupting idle CPUs is considered socially unacceptable, especially by people with battery-powered embedded systems. RCU therefore conserves energy by detecting which CPUs are idle, including tracking CPUs that have been interrupted from idle. This is a large part of the energy-efficiency requirement, so I learned of this via an irate phone call.

Because RCU avoids interrupting idle CPUs, it is illegal to execute an RCU read-side critical section on an idle CPU. (Kernels built with CONFIG_PROVE_RCU=y will splat if you try it.)

It is similarly socially unacceptable to interrupt an nohz_full CPU running in userspace. RCU must therefore track nohz_full userspace execution. RCU must therefore be able to sample state at two points in time, and be able to determine whether or not some other CPU spent any time idle and/or executing in userspace.

These energy-efficiency requirements have proven quite difficult to understand and to meet, for example, there have been more than five clean-sheet rewrites of RCU's energy-efficiency code, the last of which was finally able to demonstrate real energy savings running on real hardware [PDF]. As noted earlier, I learned of many of these requirements via angry phone calls: Flaming me on the Linux-kernel mailing list was apparently not sufficient to fully vent their ire at RCU's energy-efficiency bugs!

Scheduling-Clock Interrupts and RCU

The kernel transitions between in-kernel non-idle execution, userspace execution, and the idle loop. Depending on kernel configuration, RCU handles these states differently:

HZ Kconfig	In-Kernel	Usermode	Idle
HZ_PERIODIC	Can rely on scheduling-clock interrupt.	Can rely on scheduling-clock interrupt and its detection of interrupt from usermode.	Can rely on RCU's dyntick-idle detection.
NO_HZ_IDLE	Can rely on scheduling-clock interrupt.	Can rely on scheduling-clock interrupt and its detection of interrupt from usermode.	Can rely on RCU's dyntick-idle detection.
NO_HZ_FULL	Can only sometimes rely on scheduling-clock interrupt. In other cases, it is necessary to bound kernel execution times and/or use IPIs.	Can rely on RCU's dyntick-idle detection.	Can rely on RCU's dyntick-idle detection.

Quick Quiz:

Why can't `NO_HZ_FULL` in-kernel execution rely on the scheduling-clock interrupt, just like `HZ_PERIODIC` and `NO_HZ_IDLE` do?

Answer:

Because, as a performance optimization, `NO_HZ_FULL` does not necessarily re-enable the scheduling-clock interrupt on entry to each and every system call.

However, RCU must be reliably informed as to whether any given CPU is currently in the idle loop, and, for `NO_HZ_FULL`, also whether that CPU is executing in usermode, as discussed *earlier*. It also requires that the scheduling-clock interrupt be enabled when RCU needs it to be:

1. If a CPU is either idle or executing in usermode, and RCU believes it is non-idle, the scheduling-clock tick had better be running. Otherwise, you will get RCU CPU stall warnings. Or at best, very long (11-second) grace periods, with a pointless IPI waking the CPU from time to time.
2. If a CPU is in a portion of the kernel that executes RCU read-side critical sections, and RCU believes this CPU to be idle, you will get random memory corruption. **DON'T DO THIS!!!** This is one reason to test with lockdep, which will complain about this sort of thing.
3. If a CPU is in a portion of the kernel that is absolutely positively no-joking guaranteed to never execute any RCU read-side critical sections, and RCU believes this CPU to be idle, no problem. This sort of thing is used by some architectures for light-weight exception handlers, which can then avoid the overhead of ct_irq_enter() and ct_irq_exit() at exception entry and exit, respectively. Some go further and avoid the entireties of irq_enter() and irq_exit(). Just make very sure you are running some of your tests with `CONFIG_PROVE_RCU=y`, just in case one of your code paths was in fact joking about not doing RCU read-side critical sections.
4. If a CPU is executing in the kernel with the scheduling-clock interrupt disabled and RCU believes this CPU to be non-idle, and if the CPU goes idle (from an RCU perspective) every few jiffies, no problem. It is usually OK for there to be the occasional gap between idle periods of up to a second or so. If the gap grows too long, you get RCU CPU stall warnings.
5. If a CPU is either idle or executing in usermode, and RCU believes it to be idle, of course no problem.
6. If a CPU is executing in the kernel, the kernel code path is passing through quiescent states at a reasonable frequency (preferably about once per few jiffies, but the occasional excursion to a second or so is usually OK) and the scheduling-clock interrupt is enabled, of course no problem. If the gap between a successive pair of quiescent states grows too long, you get RCU CPU stall warnings.

Quick Quiz:

But what if my driver has a hardware interrupt handler that can run for many seconds? I cannot invoke schedule() from an hardware interrupt handler, after all!

Answer:

One approach is to do `ct_irq_exit();ct_irq_enter();` every so often. But given that long-running interrupt handlers can cause other problems, not least for response time, shouldn't you work to keep your interrupt handler's runtime within reasonable bounds?

But as long as RCU is properly informed of kernel state transitions between in-kernel execution, usermode execution, and idle, and as long as the scheduling-clock interrupt is enabled when RCU needs it to be, you can rest assured that the bugs you encounter will be in some other part of RCU or some other part of the kernel!

Memory Efficiency

Although small-memory non-realtime systems can simply use Tiny RCU, code size is only one aspect of memory efficiency. Another aspect is the size of the `rcu_head` structure used by call_rcu() and kfree_rcu(). Although this structure contains nothing more than a pair of pointers, it does appear in many RCU-protected data structures, including some that are size critical. The `page` structure is a case in point, as evidenced by the many occurrences of the `union` keyword within that structure.

This need for memory efficiency is one reason that RCU uses hand-crafted singly linked lists to track the `rcu_head` structures that are waiting for a grace period to elapse. It is also the reason why `rcu_head` structures do not contain debug information, such as fields tracking the file and line of the call_rcu() or kfree_rcu() that posted them. Although this information might appear in debug-only kernel builds at some point, in the meantime, the `->func` field will often provide the needed debug information.

However, in some cases, the need for memory efficiency leads to even more extreme measures. Returning to the `page` structure, the `rcu_head` field shares storage with a great many other structures that are used at various points in the corresponding page's lifetime. In order to correctly resolve certain race conditions, the Linux kernel's memory-management subsystem needs a particular bit to remain zero during all phases of grace-period processing, and that bit happens to map to the bottom bit of the `rcu_head` structure's `->next` field. RCU makes this guarantee as long as call_rcu() is used to post the callback, as opposed to kfree_rcu() or some future "lazy" variant of call_rcu() that might one day be created for energy-efficiency purposes.

That said, there are limits. RCU requires that the `rcu_head` structure be aligned to a two-byte boundary, and passing a misaligned `rcu_head` structure to one of the call_rcu() family of functions will result in a splat. It is therefore necessary to exercise caution when packing structures containing fields of type `rcu_head`. Why not a four-byte or even eight-byte alignment requirement? Because the m68k architecture provides only two-byte alignment, and thus acts as alignment's least common denominator.

The reason for reserving the bottom bit of pointers to `rcu_head` structures is to leave the door open to "lazy" callbacks whose invocations can safely be deferred. Deferring invocation could potentially have energy-efficiency benefits, but only if the rate of non-lazy callbacks decreases significantly for some important workload. In the meantime, reserving the bottom bit keeps this option open in case it one day becomes useful.

Performance, Scalability, Response Time, and Reliability

Expanding on the *earlier discussion*, RCU is used heavily by hot code paths in performance-critical portions of the Linux kernel's networking, security, virtualization, and scheduling code paths. RCU must therefore use efficient implementations, especially in its read-side primitives. To that end, it would be good if preemptible RCU's implementation of rcu_read_lock() could be inlined, however, doing this requires resolving `#include` issues with the `task_struct` structure.

The Linux kernel supports hardware configurations with up to 4096 CPUs, which means that RCU must be extremely scalable. Algorithms that involve frequent acquisitions of global locks or frequent atomic operations on global variables simply cannot be tolerated within the RCU implementation. RCU therefore makes heavy use of a combining tree based on the `rcu_node` structure. RCU is required to tolerate all CPUs continuously invoking any combination of RCU's runtime primitives with minimal per-operation overhead. In fact, in many cases, increasing load must *decrease* the per-operation overhead, witness the batching optimizations for synchronize_rcu(), call_rcu(), `synchronize_rcu_expedited()`, and `rcu_barrier()`. As a general rule, RCU must cheerfully accept whatever the rest of the Linux kernel decides to throw at it.

The Linux kernel is used for real-time workloads, especially in conjunction with the -rt patchset. The real-time-latency response requirements are such that the traditional approach of disabling preemption across RCU read-side critical sections is inappropriate. Kernels built with `CONFIG_PREEMPTION=y` therefore use an RCU implementation that allows RCU read-side critical sections to be preempted. This requirement made its presence known after users made it clear that an earlier real-time patch did not meet their needs, in conjunction with some RCU issues encountered by a very early version of the -rt patchset.

In addition, RCU must make do with a sub-100-microsecond real-time latency budget. In fact, on smaller systems with the -rt patchset, the Linux kernel provides sub-20-microsecond real-time latencies for the whole kernel, including RCU. RCU's scalability and latency must therefore be sufficient for these sorts of configurations. To my surprise, the sub-100-microsecond real-time latency budget applies to even the largest systems [PDF], up to and including systems with 4096 CPUs. This real-time requirement motivated the grace-period kthread, which also simplified handling of a number of race conditions.

RCU must avoid degrading real-time response for CPU-bound threads, whether executing in usermode (which is one use case for `CONFIG_NO_HZ_FULL=y`) or in the kernel. That said, CPU-bound loops in the kernel must execute cond_resched() at least once per few tens of milliseconds in order to avoid receiving an IPI from RCU.

Finally, RCU's status as a synchronization primitive means that any RCU failure can result in arbitrary memory corruption that can be extremely difficult to debug. This means that RCU must be extremely reliable, which in practice also means that RCU must have an aggressive stress-test suite. This stress-test suite is called `rcutorture`.

Although the need for `rcutorture` was no surprise, the current immense popularity of the Linux kernel is posing interesting—and perhaps unprecedented—validation challenges. To see this, keep in mind that there are well over one billion instances of the Linux kernel running today, given Android smartphones, Linux-powered televisions, and servers. This number can be expected to increase sharply with the advent of the celebrated Internet of Things.

Suppose that RCU contains a race condition that manifests on average once per million years of runtime. This bug will be occurring about three times per *day* across the installed base. RCU could simply hide behind hardware error rates, given that no one should really expect their smartphone to last for a million years. However, anyone taking too much comfort from this thought should consider the fact that in most jurisdictions, a successful multi-year test of a given mechanism, which might include a Linux kernel, suffices for a number of types of safety-critical certifications. In fact, rumor has it that the Linux kernel is already being used in production for safety-critical applications. I don't know about you, but I would feel quite bad if a bug in RCU killed someone. Which might explain my recent focus on validation and verification.

Other RCU Flavors

One of the more surprising things about RCU is that there are now no fewer than five *flavors*, or API families. In addition, the primary flavor that has been the sole focus up to this point has two different implementations, non-preemptible and preemptible. The other four flavors are listed below, with requirements for each described in a separate section.

1. *Bottom-Half Flavor (Historical)*
2. *Sched Flavor (Historical)*
3. *Sleepable RCU*
4. *Tasks RCU*

Bottom-Half Flavor (Historical)

The RCU-bh flavor of RCU has since been expressed in terms of the other RCU flavors as part of a consolidation of the three flavors into a single flavor. The read-side API remains, and continues to disable softirq and to be accounted for by lockdep. Much of the material in this section is therefore strictly historical in nature.

The softirq-disable (AKA "bottom-half", hence the "_bh" abbreviations) flavor of RCU, or *RCU-bh*, was developed by Dipankar Sarma to provide a flavor of RCU that could withstand the network-based denial-of-service attacks researched by Robert Olsson. These attacks placed so much networking load on the system that some of the CPUs never exited softirq execution, which in turn prevented those CPUs from ever executing a context switch, which, in the RCU implementation of that time, prevented grace periods from ever ending. The result was an out-of-memory condition and a system hang.

The solution was the creation of RCU-bh, which does local_bh_disable() across its read-side critical sections, and which uses the transition from one type of softirq processing to another as a quiescent state in addition to context switch, idle, user mode, and offline. This means that RCU-bh grace periods can complete even when some of the CPUs execute in softirq indefinitely, thus allowing algorithms based on RCU-bh to withstand network-based denial-of-service attacks.

Because `rcu_read_lock_bh()` and `rcu_read_unlock_bh()` disable and re-enable softirq handlers, any attempt to start a softirq handlers during the RCU-bh read-side critical section will be deferred. In this case, `rcu_read_unlock_bh()` will invoke softirq processing, which can take considerable time. One can of course argue that this softirq overhead should be associated with the code following the RCU-bh read-side critical section rather than `rcu_read_unlock_bh()`, but the fact is that most profiling tools cannot be expected to make this sort of fine distinction. For example, suppose that a three-millisecond-long RCU-bh read-side critical section executes during a time of heavy networking load. There will very likely be an attempt to invoke at least one softirq handler during that three milliseconds, but any such invocation will be delayed until the time of the *rcu_read_unlock_bh()*. This can of course make it appear at first glance as if *rcu_read_unlock_bh()* was executing very slowly.

The RCU-bh API includes *rcu_read_lock_bh()*, *rcu_read_unlock_bh()*, *rcu_dereference_bh()*, *rcu_dereference_bh_check()*, and *rcu_read_lock_bh_held()*. However, the old RCU-bh update-side APIs are now gone, replaced by synchronize_rcu(), *synchronize_rcu_expedited()*, call_rcu(), and *rcu_barrier()*. In addition, anything that disables bottom halves also marks an RCU-bh read-side critical section, including local_bh_disable() and local_bh_enable(), local_irq_save() and local_irq_restore(), and so on.

Sched Flavor (Historical)

The RCU-sched flavor of RCU has since been expressed in terms of the other RCU flavors as part of a consolidation of the three flavors into a single flavor. The read-side API remains, and continues to disable preemption and to be accounted for by lockdep. Much of the material in this section is therefore strictly historical in nature.

Before preemptible RCU, waiting for an RCU grace period had the side effect of also waiting for all pre-existing interrupt and NMI handlers. However, there are legitimate preemptible-RCU implementations that do not have this property, given that any point in the code outside of an RCU read-side critical section can be a quiescent state. Therefore, *RCU-sched* was created, which follows "classic" RCU in that an RCU-sched grace period waits for pre-existing interrupt and NMI handlers. In kernels built with `CONFIG_PREEMPTION=n`, the RCU and RCU-sched APIs have identical implementations, while kernels built with `CONFIG_PREEMPTION=y` provide a separate implementation for each.

Note well that in `CONFIG_PREEMPTION=y` kernels, *rcu_read_lock_sched()* and *rcu_read_unlock_sched()* disable and re-enable preemption, respectively. This means that if there was a preemption attempt during the RCU-sched read-side critical section, *rcu_read_unlock_sched()* will enter the scheduler, with all the latency and overhead entailed. Just as with *rcu_read_unlock_bh()*, this can make it look as if *rcu_read_unlock_sched()* was executing very slowly. However, the highest-priority task won't be preempted, so that task will enjoy low-overhead *rcu_read_unlock_sched()* invocations.

The RCU-sched API includes *rcu_read_lock_sched()*, *rcu_read_unlock_sched()*, rcu_read_lock_sched_notrace(), rcu_read_unlock_sched_notrace(), *rcu_dereference_sched()*, *rcu_dereference_sched_check()*, and rcu_read_lock_sched_held(). However, the old RCU-sched update-side APIs are now gone, replaced by synchronize_rcu(), *synchronize_rcu_expedited()*, call_rcu(), and *rcu_barrier()*. In addition, anything that disables preemption also marks an RCU-sched read-side critical section, including preempt_disable() and preempt_enable(), local_irq_save() and local_irq_restore(), and so on.

Sleepable RCU

For well over a decade, someone saying "I need to block within an RCU read-side critical section" was a reliable indication that this someone did not understand RCU. After all, if you are always blocking in an RCU read-side critical section, you can probably afford to use a higher-overhead synchronization mechanism. However, that changed with the advent of the Linux kernel's notifiers, whose RCU read-side critical sections almost never sleep, but sometimes need to. This resulted in the introduction of sleepable RCU, or *SRCU*.

SRCU allows different domains to be defined, with each such domain defined by an instance of an `srcu_struct` structure. A pointer to this structure must be passed in to each SRCU function, for example, `synchronize_srcu(&ss)`, where `ss` is the `srcu_struct` structure. The key benefit of these domains is that a slow SRCU reader in one domain does not delay an SRCU grace period in some other domain. That said, one consequence of these domains is that read-side code must pass a "cookie" from *srcu_read_lock()* to *srcu_read_unlock()*, for example, as follows:

```
int idx;

idx = srcu_read_lock(&ss);
do_something();
srcu_read_unlock(&ss, idx);
```

As noted above, it is legal to block within SRCU read-side critical sections, however, with great power comes great responsibility. If you block forever in one of a given domain's SRCU read-side critical sections, then that domain's grace periods will also be blocked forever. Of course, one good way to block forever is to deadlock, which can happen if any operation in a given domain's SRCU read-side critical section can wait, either directly or indirectly, for that domain's grace period to elapse. For example, this results in a self-deadlock:

```
1 int idx;
2
3 idx = srcu_read_lock(&ss);
4 do_something();
5 synchronize_srcu(&ss);
6 srcu_read_unlock(&ss, idx);
```

However, if line 5 acquired a mutex that was held across a synchronize_srcu() for domain `ss`, deadlock would still be possible. Furthermore, if line 5 acquired a mutex that was held across a synchronize_srcu() for some other domain `ss1`, and if an `ss1`-domain SRCU read-side critical section acquired another mutex that was held across as `ss`-domain synchronize_srcu(), deadlock would again be possible. Such a deadlock cycle could extend across an arbitrarily large number of different SRCU domains. Again, with great power comes great responsibility.

Unlike the other RCU flavors, SRCU read-side critical sections can run on idle and even offline CPUs. This ability requires that *srcu_read_lock()* and *srcu_read_unlock()* contain memory barriers, which means that SRCU readers will run a bit slower than would RCU readers. It also motivates the *smp_mb__after_srcu_read_unlock()* API, which, in combination with *srcu_read_unlock()*, guarantees a full memory barrier.

Also unlike other RCU flavors, synchronize_srcu() may **not** be invoked from CPU-hotplug notifiers, due to the fact that SRCU grace periods make use of timers and the possibility of timers being temporarily "stranded" on the outgoing CPU. This stranding of timers means that timers posted to the outgoing CPU will not fire until late in the CPU-hotplug process. The problem is that if a notifier is waiting on an SRCU grace period, that grace period is waiting on a timer, and that timer is stranded on the outgoing CPU, then the notifier will never be awakened, in other words, deadlock has occurred. This same situation of course also prohibits *srcu_barrier()* from being invoked from CPU-hotplug notifiers.

SRCU also differs from other RCU flavors in that SRCU's expedited and non-expedited grace periods are implemented by the same mechanism. This means that in the current SRCU implementation, expediting a future grace period has the side effect of expediting all prior grace periods that have not yet completed. (But please note that this is a property of the current implementation, not necessarily of future implementations.) In addition, if SRCU has been idle for longer than the interval specified by the `srcutree.exp_holdoff` kernel boot parameter (25 microseconds by default), and if a synchronize_srcu() invocation ends this idle period, that invocation will be automatically expedited.

As of v4.12, SRCU's callbacks are maintained per-CPU, eliminating a locking bottleneck present in prior kernel versions. Although this will allow users to put much heavier stress on *call_srcu()*, it is important to note that SRCU does not yet take any special steps to deal with callback flooding. So if you are posting (say) 10,000 SRCU callbacks per second per CPU, you are probably totally OK, but if you intend to post (say) 1,000,000 SRCU callbacks per second per CPU, please run some tests first. SRCU just might need a few adjustment to deal with that sort of load. Of course, your mileage may vary based on the speed of your CPUs and the size of your memory.

The SRCU API includes *srcu_read_lock()*, *srcu_read_unlock()*, *srcu_dereference()*, *srcu_dereference_check()*, synchronize_srcu(), *synchronize_srcu_expedited()*, *call_srcu()*, *srcu_barrier()*, and *srcu_read_lock_held()*. It also includes DEFINE_SRCU(), DEFINE_STATIC_SRCU(), and *init_srcu_struct()* APIs for defining and initializing `srcu_struct` structures.

More recently, the SRCU API has added polling interfaces:

1. *start_poll_synchronize_srcu()* returns a cookie identifying the completion of a future SRCU grace period and ensures that this grace period will be started.
2. *poll_state_synchronize_srcu()* returns `true` iff the specified cookie corresponds to an already-completed SRCU grace period.
3. *get_state_synchronize_srcu()* returns a cookie just like *start_poll_synchronize_srcu()* does, but differs in that it does nothing to ensure that any future SRCU grace period will be started.

These functions are used to avoid unnecessary SRCU grace periods in certain types of buffer-cache algorithms having multi-stage age-out mechanisms. The idea is that by the time the block has aged completely from the cache, an SRCU grace period will be very likely to have elapsed.

Tasks RCU

Some forms of tracing use "trampolines" to handle the binary rewriting required to install different types of probes. It would be good to be able to free old trampolines, which sounds like a job for some form of RCU. However, because it is necessary to be able to install a trace anywhere in the code, it is not possible to use read-side markers such as rcu_read_lock() and rcu_read_unlock(). In addition, it does not work to have these markers in the trampoline itself, because there would need to be instructions following rcu_read_unlock(). Although synchronize_rcu() would guarantee that execution reached the rcu_read_unlock(), it would not be able to guarantee that execution had completely left the trampoline. Worse yet, in some situations the trampoline's protection must extend a few instructions *prior* to execution reaching the trampoline. For example, these few instructions might calculate the address of the trampoline, so that entering the trampoline would be pre-ordained a surprisingly long time before execution actually reached the trampoline itself.

The solution, in the form of Tasks RCU, is to have implicit read-side critical sections that are delimited by voluntary context switches, that is, calls to schedule(), cond_resched(), and *synchronize_rcu_tasks()*. In addition, transitions to and from userspace execution also delimit tasks-RCU read-side critical sections.

The tasks-RCU API is quite compact, consisting only of *call_rcu_tasks()*, *synchronize_rcu_tasks()*, and *rcu_barrier_tasks()*. In `CONFIG_PREEMPTION=n` kernels, trampolines cannot be preempted, so these APIs map to call_rcu(), synchronize_rcu(), and *rcu_barrier()*, respectively. In `CONFIG_PREEMPTION=y` kernels, trampolines can be preempted, and these three APIs are therefore implemented by separate functions that check for voluntary context switches.

Tasks Rude RCU

Some forms of tracing need to wait for all preemption-disabled regions of code running on any online CPU, including those executed when RCU is not watching. This means that synchronize_rcu() is insufficient, and Tasks Rude RCU must be used instead. This flavor of RCU does its work by forcing a workqueue to be scheduled on each online CPU, hence the "Rude" moniker. And this operation is considered to be quite rude by real-time workloads that don't want their `nohz_full` CPUs receiving IPIs and by battery-powered systems that don't want their idle CPUs to be awakened.

The tasks-rude-RCU API is also reader-marking-free and thus quite compact, consisting of *call_rcu_tasks_rude()*, *synchronize_rcu_tasks_rude()*, and *rcu_barrier_tasks_rude()*.

Tasks Trace RCU

Some forms of tracing need to sleep in readers, but cannot tolerate SRCU's read-side overhead, which includes a full memory barrier in both *srcu_read_lock()* and *srcu_read_unlock()*. This need is handled by a Tasks Trace RCU that uses scheduler locking and IPIs to synchronize with readers. Real-time systems that cannot tolerate IPIs may build their kernels with `CONFIG_TASKS_TRACE_RCU_READ_MB=y`, which avoids the IPIs at the expense of adding full memory barriers to the read-side primitives.

The tasks-trace-RCU API is also reasonably compact, consisting of *rcu_read_lock_trace()*, *rcu_read_unlock_trace()*, rcu_read_lock_trace_held(), *call_rcu_tasks_trace()*, *synchronize_rcu_tasks_trace()*, and *rcu_barrier_tasks_trace()*.

Possible Future Changes

One of the tricks that RCU uses to attain update-side scalability is to increase grace-period latency with increasing numbers of CPUs. If this becomes a serious problem, it will be necessary to rework the grace-period state machine so as to avoid the need for the additional latency.

RCU disables CPU hotplug in a few places, perhaps most notably in the *rcu_barrier()* operations. If there is a strong reason to use *rcu_barrier()* in CPU-hotplug notifiers, it will be necessary to avoid disabling CPU hotplug. This would introduce some complexity, so there had better be a *very* good reason.

The tradeoff between grace-period latency on the one hand and interruptions of other CPUs on the other hand may need to be re-examined. The desire is of course for zero grace-period latency as well as zero interprocessor interrupts undertaken during an expedited grace period operation. While this ideal is unlikely to be achievable, it is quite possible that further improvements can be made.

The multiprocessor implementations of RCU use a combining tree that groups CPUs so as to reduce lock contention and increase cache locality. However, this combining tree does not spread its memory across NUMA nodes nor does it align the CPU groups with hardware features such as sockets or cores. Such spreading and alignment is currently believed to be unnecessary because the hotpath read-side primitives do not access the combining tree, nor does call_rcu() in the common case. If you believe that your architecture needs such spreading and alignment, then your architecture should also benefit from the `rcutree.rcu_fanout_leaf` boot parameter, which can be set to the number of CPUs in a socket, NUMA node, or whatever. If the number of CPUs is too large, use a fraction of the number of CPUs. If the number of CPUs is a large prime number, well, that certainly is an "interesting" architectural choice! More flexible arrangements might be considered, but only if `rcutree.rcu_fanout_leaf` has proven inadequate, and only if the inadequacy has been demonstrated by a carefully run and realistic system-level workload.

Please note that arrangements that require RCU to remap CPU numbers will require extremely good demonstration of need and full exploration of alternatives.

RCU's various kthreads are reasonably recent additions. It is quite likely that adjustments will be required to more gracefully handle extreme loads. It might also be necessary to be able to relate CPU utilization by RCU's kthreads and softirq handlers to the code that instigated this CPU utilization. For example, RCU callback overhead might be charged back to the originating call_rcu() instance, though probably not in production kernels.

Additional work may be required to provide reasonable forward-progress guarantees under heavy load for grace periods and for callback invocation.

Summary

This document has presented more than two decade's worth of RCU requirements. Given that the requirements keep changing, this will not be the last word on this subject, but at least it serves to get an important subset of the requirements set forth.

Acknowledgments

I am grateful to Steven Rostedt, Lai Jiangshan, Ingo Molnar, Oleg Nesterov, Borislav Petkov, Peter Zijlstra, Boqun Feng, and Andy Lutomirski for their help in rendering this article human readable, and to Michelle Rankin for her support of this effort. Other contributions are acknowledged in the Linux kernel's git archive.

4.5.18 A Tour Through TREE_RCU's Data Structures [LWN.net]

December 18, 2016

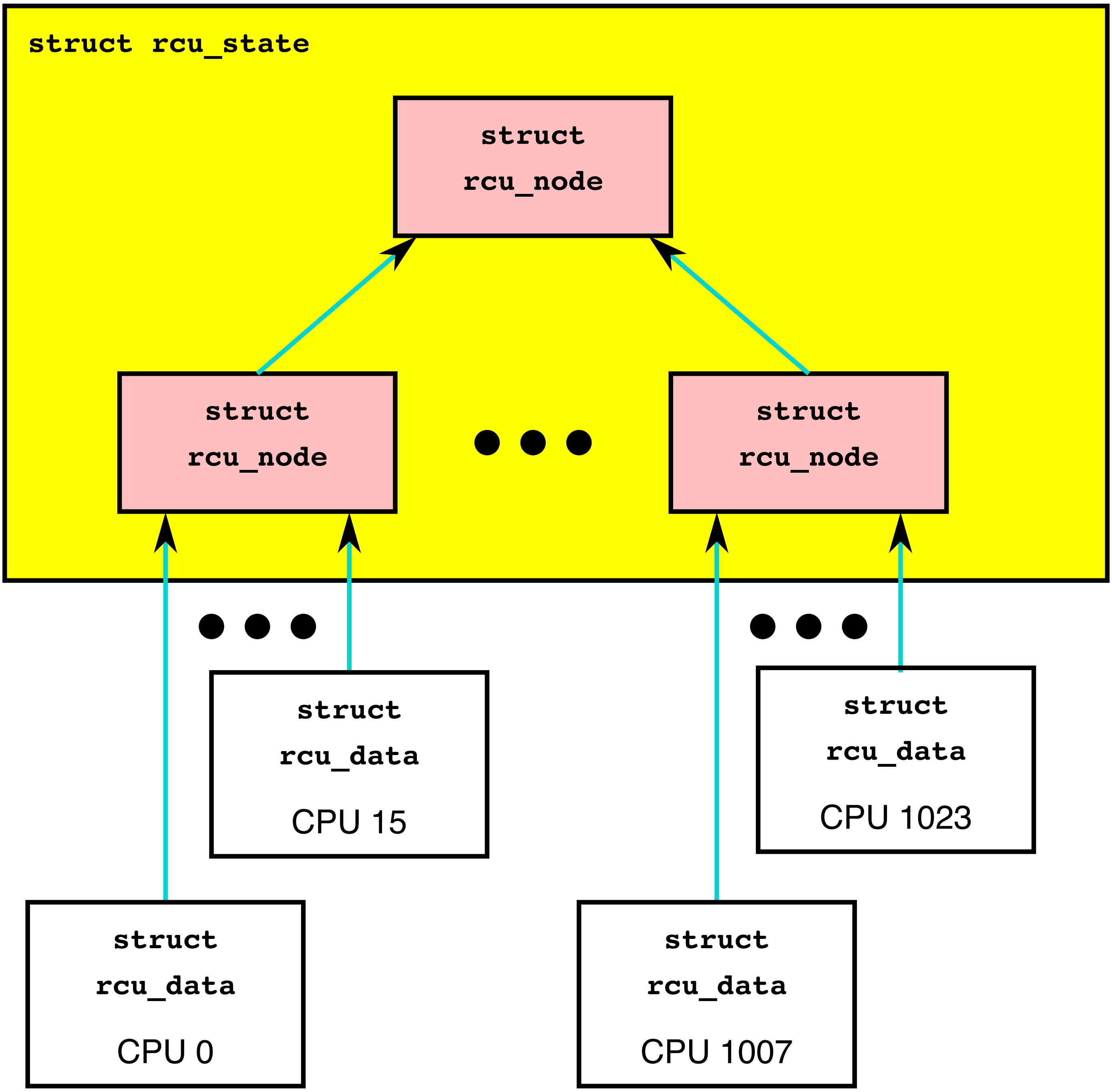
This article was contributed by Paul E. McKenney

Introduction

This document describes RCU's major data structures and their relationship to each other.

Data-Structure Relationships

RCU is for all intents and purposes a large state machine, and its data structures maintain the state in such a way as to allow RCU readers to execute extremely quickly, while also processing the RCU grace periods requested by updaters in an efficient and extremely scalable fashion. The efficiency and scalability of RCU updaters is provided primarily by a combining tree, as shown below:



This diagram shows an enclosing rcu_state structure containing a tree of rcu_node structures. Each leaf node of the rcu_node tree has up to 16 rcu_data structures associated with it, so that there are NR_CPUS number of rcu_data structures, one for each possible CPU. This structure is adjusted at boot time, if needed, to handle the common case where nr_cpu_ids is much less than NR_CPUs. For example, a number of Linux distributions set NR_CPUs=4096, which results in a three-level rcu_node tree. If the actual hardware has only 16 CPUs, RCU will adjust itself at boot time, resulting in an rcu_node tree with only a single node.

The purpose of this combining tree is to allow per-CPU events such as quiescent states, dyntick-idle transitions, and CPU hotplug operations to be processed efficiently and scalably. Quiescent states are recorded by the per-CPU rcu_data structures, and other events are recorded by the leaf-level rcu_node structures. All of these events are combined at each level of the tree until finally grace periods are completed at the tree's root rcu_node structure. A grace period can be completed at the root once every CPU (or, in the case of CONFIG_PREEMPT_RCU, task) has passed through a quiescent state. Once a grace period has completed, record of that fact is propagated back down the tree.

As can be seen from the diagram, on a 64-bit system a two-level tree with 64 leaves can accommodate 1,024 CPUs, with a fanout of 64 at the root and a fanout of 16 at the leaves.

Quick Quiz:

Why isn't the fanout at the leaves also 64?

Answer:

Because there are more types of events that affect the leaf-level rcu_node structures than further up the tree. Therefore, if the leaf rcu_node structures have fanout of 64, the contention on these structures' ->structures becomes excessive. Experimentation on a wide variety of systems has shown that a fanout of 16 works well for the leaves of the rcu_node tree.

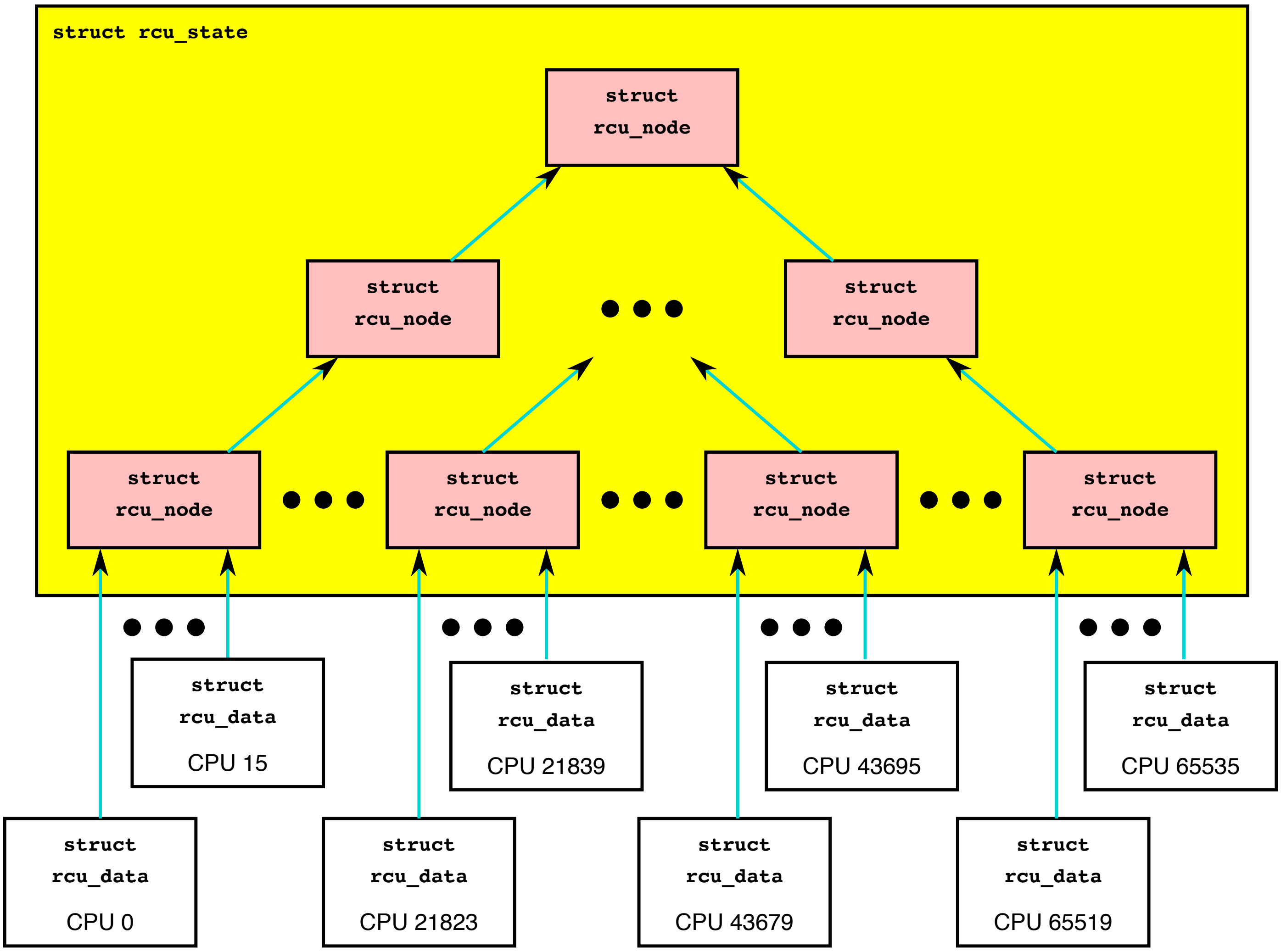
Of course, further experience with systems having hundreds or thousands of CPUs may demonstrate that the fanout for the non-leaf rcu_node structures must also be reduced. Such reduction can be easily carried out when and if it proves necessary. In the meantime, if you are using such a system and running into contention problems on the non-leaf rcu_node structures, you may use the CONFIG_RCU_FANOUT kernel configuration parameter to reduce the non-leaf fanout as needed.

Kernels built for systems with strong NUMA characteristics might also need to adjust CONFIG_RCU_FANOUT so that the domains of the rcu_node structures align with hardware boundaries. However, there has thus far been no need for this.

If your system has more than 1,024 CPUs (or more than 512 CPUs on a 32-bit system), then RCU will automatically add more levels to the tree. For example, if you are crazy enough to build a 64-bit system with 65,536 CPUs, RCU would configure the rcu_node tree as follows:

RCU currently permits up to a four-level tree, which on a 64-bit system accommodates up to 4,194,304 CPUs, though only a mere 524,288 CPUs for 32-bit systems. On the other hand, you can set both CONFIG_RCU_FANOUT and CONFIG_RCU_FANOUT_LEAF to be as small as 2, which would result in a 16-CPU test using a 4-level tree. This can be useful for testing large-system capabilities on small test machines.

This multi-level combining tree allows us to get most of the performance and scalability benefits of partitioning, even though RCU grace-period detection is inherently a global operation. The trick here is that only the last CPU to report a quiescent state into a given rcu_node structure need advance to the rcu_node structure at the next level up the tree. This means that at the



struct rcu_state
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_data
CPU 15
struct rcu_data
CPU 21839
struct rcu_data
CPU 43695
struct rcu_data
CPU 65535
struct rcu_data
CPU 0
struct rcu_data
CPU 21823
struct rcu_data
CPU 43679
struct rcu_data
CPU 65519

leaf-level rcu_node structure, only one access out of sixteen will progress up the tree. For the internal rcu_node structures, the situation is even more extreme: Only one access out of sixty-four will progress up the tree. Because the vast majority of the CPUs do not progress up the tree, the lock contention remains roughly constant up the tree. No matter how many CPUs there are in the system, at most 64 quiescent-state reports per grace period will progress all the way to the root rcu_node structure, thus ensuring that the lock contention on that root rcu_node structure remains acceptably low.

In effect, the combining tree acts like a big shock absorber, keeping lock contention under control at all tree levels regardless of the level of loading on the system.

RCU updaters wait for normal grace periods by registering RCU callbacks, either directly via call_rcu() or indirectly via synchronize_rcu() and friends. RCU callbacks are represented by rcu_head structures, which are queued on rcu_data structures while they are waiting for a grace period to elapse, as shown in the following figure:

This figure shows how TREE_RCU's and PREEMPT_RCU's major data structures are related. Lesser data structures will be introduced with the algorithms that make use of them.

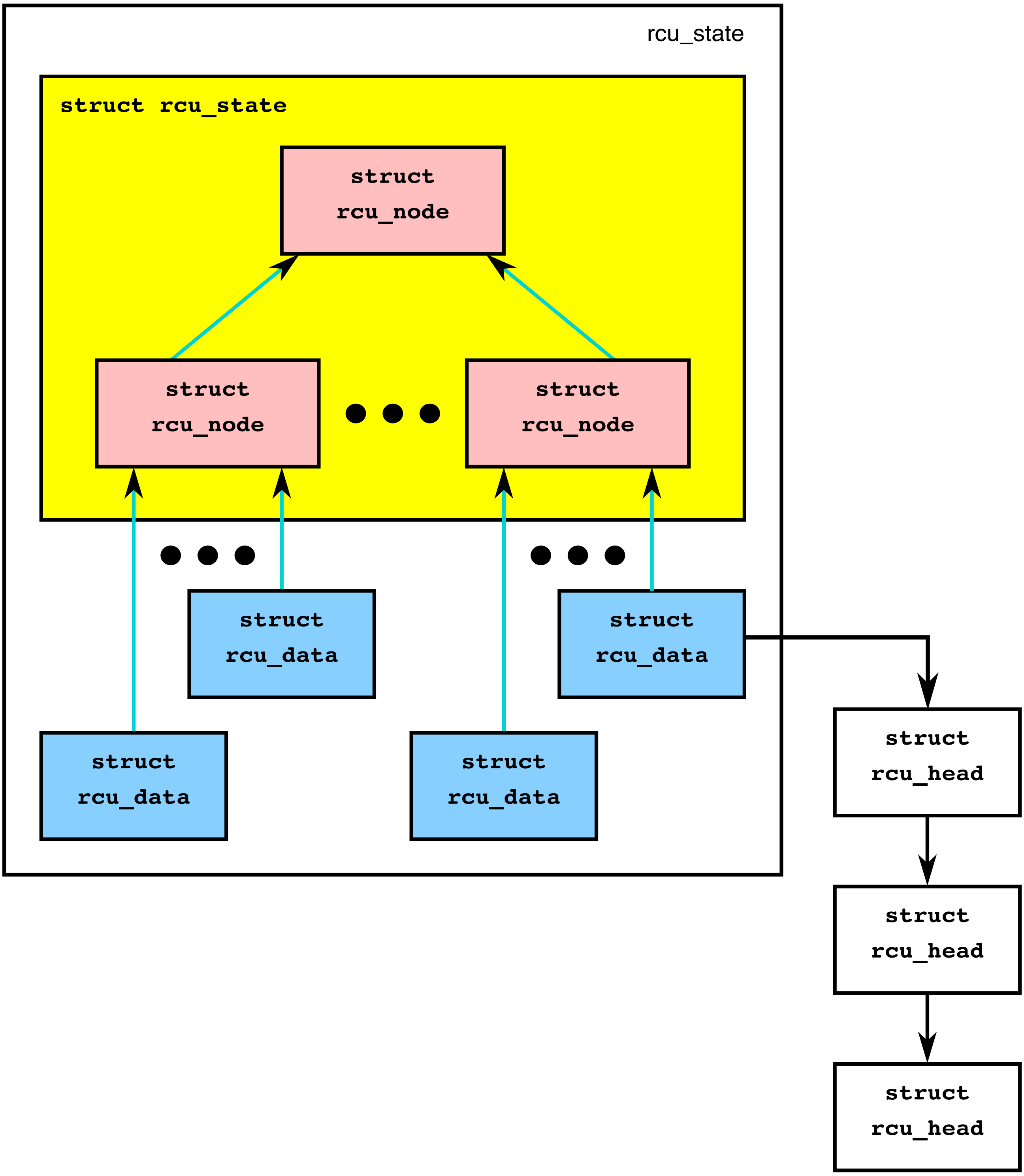
Note that each of the data structures in the above figure has its own synchronization:

1. Each rcu_state structures has a lock and a mutex, and some fields are protected by the corresponding root rcu_node structure's lock.
2. Each rcu_node structure has a spinlock.
3. The fields in rcu_data are private to the corresponding CPU, although a few can be read and written by other CPUs.

It is important to note that different data structures can have very different ideas about the state of RCU at any given time. For but one example, awareness of the start or end of a given RCU grace period propagates slowly through the data structures. This slow propagation is absolutely necessary for RCU to have good read-side performance. If this balkanized implementation seems foreign to you, one useful trick is to consider each instance of these data structures to be a different person, each having the usual slightly different view of reality.

The general role of each of these data structures is as follows:

1. rcu_state: This structure forms the interconnection between the rcu_node and rcu_data structures, tracks grace periods, serves as short-term repository for callbacks orphaned by CPU-hotplug events, maintains rcu_barrier() state, tracks expedited grace-period state, and maintains state used to force quiescent states when grace periods extend too long,
2. rcu_node: This structure forms the combining tree that propagates quiescent-state information from the leaves to the root, and also propagates grace-period information from the root to the leaves. It provides local copies of the grace-period state in order to allow this information to be accessed in a synchronized manner without suffering the scalability limitations that would otherwise be imposed by global locking. In CONFIG_PREEMPT_RCU kernels, it manages the lists of tasks that have blocked while in their current RCU read-side critical section. In CONFIG_PREEMPT_RCU with CONFIG_RCU_BOOST, it manages the per-rcu_node priority-boosting kernel threads (kthreads) and state. Finally, it records CPU-hotplug state in order to determine which CPUs should be ignored during a given grace period.
3. rcu_data: This per-CPU structure is the focus of quiescent-state detection and RCU callback queuing. It also tracks its relationship to the corresponding leaf rcu_node structure to allow more-efficient propagation of quiescent states up the rcu_node combining tree.


rcu_state
struct rcu_state
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_data
struct rcu_data
struct rcu_data
struct rcu_data
struct rcu_head
struct rcu_head
struct rcu_head

Like the rcu_node structure, it provides a local copy of the grace-period information to allow for-free synchronized access to this information from the corresponding CPU. Finally, this structure records past dyntick-idle state for the corresponding CPU and also tracks statistics.

4. rcu_head: This structure represents RCU callbacks, and is the only structure allocated and managed by RCU users. The rcu_head structure is normally embedded within the RCU-protected data structure.

If all you wanted from this article was a general notion of how RCU's data structures are related, you are done. Otherwise, each of the following sections give more details on the rcu_state, rcu_node and rcu_data data structures.

The rcu_state Structure

The rcu_state structure is the base structure that represents the state of RCU in the system. This structure forms the interconnection between the rcu_node and rcu_data structures, tracks grace periods, contains the lock used to synchronize with CPU-hotplug events, and maintains state used to force quiescent states when grace periods extend too long,

A few of the rcu_state structure's fields are discussed, singly and in groups, in the following sections. The more specialized fields are covered in the discussion of their use.

Relationship to rcu_node and rcu_data Structures

This portion of the rcu_state structure is declared as follows:

```
struct rcu_node node[NUM_RCU_NODES];
struct rcu_node *level[NUM_RCU_LVLS + 1];
struct rcu_data __percpu *rda;
```

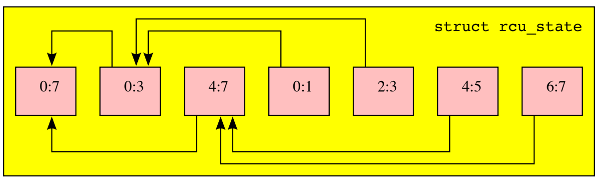
Quick Quiz:

Wait a minute! You said that the rcu_node structures formed a tree, but they are declared as a flat array! What gives?

Answer:

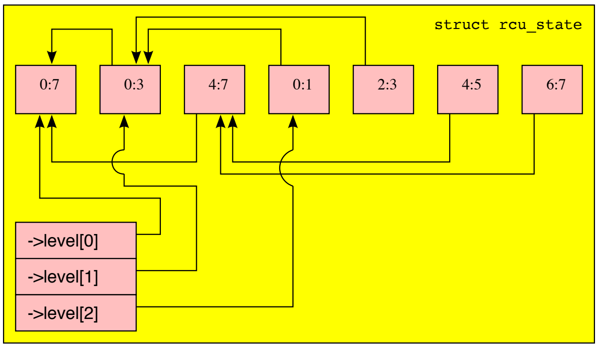
The tree is laid out in the array. The first node In the array is the head, the next set of nodes in the array are children of the head node, and so on until the last set of nodes in the array are the leaves. See the following diagrams to see how this works.

The rcu_node tree is embedded into the ->node[] array as shown in the following figure:



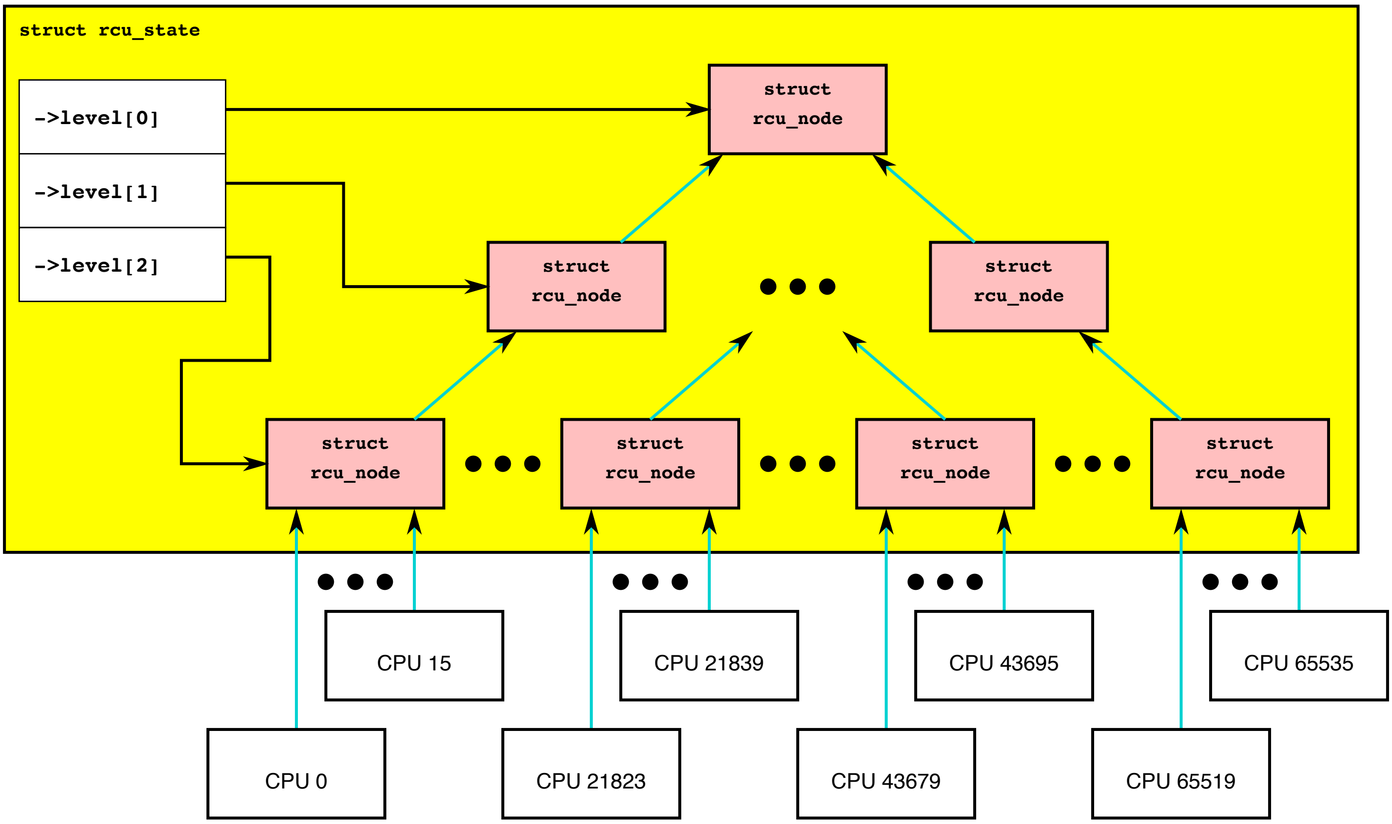
One interesting consequence of this mapping is that a breadth-first traversal of the tree is implemented as a simple linear scan of the array, which is in fact what the rcu_for_each_node_breadth_first() macro does. This macro is used at the beginning and ends of grace periods.

Each entry of the ->level array references the first rcu_node structure on the corresponding level of the tree, for example, as shown below:



The zeroth element of the array references the root rcu_node structure, the first element references the first child of the root rcu_node, and finally the second element references the first leaf rcu_node structure.

For whatever it is worth, if you draw the tree to be tree-shaped rather than array-shaped, it is easy to draw a planar representation:



Finally, the ->rda field references a per-CPU pointer to the corresponding CPU's rcu_data structure.

All of these fields are constant once initialization is complete, and therefore need no protection.

Grace-Period Tracking

This portion of the rcu_state structure is declared as follows:

```
1   unsigned long gp_seq;
```

RCU grace periods are numbered, and the ->gp_seq field contains the current grace-period sequence number. The bottom two bits are the state of the current grace period, which can be zero for not yet started or one for in progress. In other words, if the bottom two bits of ->gp_seq are zero, then RCU is idle. Any other value in the bottom two bits indicates that something is broken. This field is protected by the root rcu_node structure's ->lock field.

There are ->gp_seq fields in the rcu_node and rcu_data structures as well. The fields in the rcu_state structure represent the most current value, and those of the other structures are compared in order to detect the beginnings and ends of grace periods in a distributed fashion. The values flow from rcu_state to rcu_node (down the tree from the root to the leaves) to rcu_data.

Miscellaneous

This portion of the rcu_state structure is declared as follows:

```
1   unsigned long gp_max;
2   char abbr;
3   char *name;
```

The ->gp_max field tracks the duration of the longest grace period in jiffies. It is protected by the root rcu_node's ->lock.

The ->name and ->abbr fields distinguish between preemptible RCU ("rcu_preempt" and "p") and non-preemptible RCU ("rcu_sched" and "s"). These fields are used for diagnostic and tracing purposes.

The rcu_node Structure

The rcu_node structures form the combining tree that propagates quiescent-state information from the leaves to the root and also that propagates grace-period information from the root down to the leaves. They provides local copies of the grace-period state in order to allow this information to be accessed in a synchronized manner without suffering the scalability limitations that would otherwise be imposed by global locking. In CONFIG_PREEMPT_RCU kernels, they manage the lists of tasks that have blocked while in their current RCU read-side critical section. In CONFIG_PREEMPT_RCU with CONFIG_RCU_BOOST, they manage the per-rcu_node priority-boosting kernel threads (kthreads) and state. Finally, they record CPU-hotplug state in order to determine which CPUs should be ignored during a given grace period.

The rcu_node structure's fields are discussed, singly and in groups, in the following sections.

Connection to Combining Tree

This portion of the rcu_node structure is declared as follows:

```
struct rcu_node *parent;
u8 level;
u8 grpnum;
unsigned long grpmask;
int grplo;
int grphi;
```

The ->parent pointer references the rcu_node one level up in the tree, and is NULL for the root rcu_node. The RCU implementation makes heavy use of this field to push quiescent states up the tree. The ->level field gives the level in the tree, with the root being at level zero, its children at level one, and so on. The ->grpnum field gives this node's position within the children of its parent, so this number can range between 0 and 31 on 32-bit systems and between 0 and 63 on 64-bit systems. The ->level and ->grpnum fields are used only during initialization and for tracing. The ->grpmask field is the bitmask counterpart of ->grpnum, and therefore always has exactly one bit set. This mask is used to clear the bit corresponding to this rcu_node structure in its parent's bitmasks, which are described later. Finally, the ->grplo and ->grphi fields contain the lowest and highest numbered CPU served by this rcu_node structure, respectively.

All of these fields are constant, and thus do not require any synchronization.

Synchronization

This field of the rcu_node structure is declared as follows:

```
raw_spinlock_t lock;
```

This field is used to protect the remaining fields in this structure, unless otherwise stated. That said, all of the fields in this structure can be accessed without locking for tracing purposes. Yes, this can result in confusing traces, but better some tracing confusion than to be heisenbugged out of existence.

Grace-Period Tracking

This portion of the rcu_node structure is declared as follows:

```
unsigned long gp_seq;
unsigned long gp_seq_needed;
```

The rcu_node structures' ->gp_seq fields are the counterparts of the field of the same name in the rcu_state structure. They each may lag up to one step behind their rcu_state counterpart. If the bottom two bits of a given rcu_node structure's ->gp_seq field is zero, then this rcu_node structure believes that RCU is idle.

The >gp_seq field of each rcu_node structure is updated at the beginning and the end of each grace period.

The `->gp_seq_needed` fields record the furthest-in-the-future grace period request seen by the corresponding `rcu_node` structure. The request is considered fulfilled when the value of the `->gp_seq` field equals or exceeds that of the `->gp_seq_needed` field.

Quick Quiz:

Suppose that this `rcu_node` structure doesn't see a request for a very long time. Won't wrapping of the `->gp_seq` field cause problems?

Answer:

No, because if the `->gp_seq_needed` field lags behind the `->gp_seq` field, the `->gp_seq_needed` field will be updated at the end of the grace period. Modulo-arithmetic comparisons therefore will always get the correct answer, even with wrapping.

Quiescent-State Tracking

These fields manage the propagation of quiescent states up the combining tree.

This portion of the `rcu_node` structure has fields as follows:

```
unsigned long qsmask;
unsigned long expmask;
unsigned long qsmaskinit;
unsigned long expmaskinit;
```

The `->qsmask` field tracks which of this `rcu_node` structure's children still need to report quiescent states for the current normal grace period. Such children will have a value of 1 in their corresponding bit. Note that the leaf `rcu_node` structures should be thought of as having `rcu_data` structures as their children. Similarly, the `->expmask` field tracks which of this `rcu_node` structure's children still need to report quiescent states for the current expedited grace period. An expedited grace period has the same conceptual properties as a normal grace period, but the expedited implementation accepts extreme CPU overhead to obtain much lower grace-period latency, for example, consuming a few tens of microseconds worth of CPU time to reduce grace-period duration from milliseconds to tens of microseconds. The `->qsmaskinit` field tracks which of this `rcu_node` structure's children cover for at least one online CPU. This mask is used to initialize `->qsmask`, and `->expmaskinit` is used to initialize `->expmask` and the beginning of the normal and expedited grace periods, respectively.

Quick Quiz:

Why are these bitmasks protected by locking? Come on, haven't you heard of atomic instructions???

Answer:

Lockless grace-period computation! Such a tantalizing possibility! But consider the following sequence of events:

1. CPU 0 has been in dyntick-idle mode for quite some time. When it wakes up, it notices that the current RCU grace period needs it to report in, so it sets a flag where the scheduling clock interrupt will find it.
2. Meanwhile, CPU 1 is running `force_quiescent_state()`, and notices that CPU 0 has been in dyntick idle mode, which qualifies as an extended quiescent state.
3. CPU 0's scheduling clock interrupt fires in the middle of an RCU read-side critical section, and notices that the RCU core needs something, so commences RCU softirq processing.
4. CPU 0's softirq handler executes and is just about ready to report its quiescent state up the `rcu_node` tree.
5. But CPU 1 beats it to the punch, completing the current grace period and starting a new one.
6. CPU 0 now reports its quiescent state for the wrong grace period. That grace period might now end before the RCU read-side critical section. If that happens, disaster will ensue.

So the locking is absolutely required in order to coordinate clearing of the bits with updating of the grace-period sequence number in `->gp_seq`.

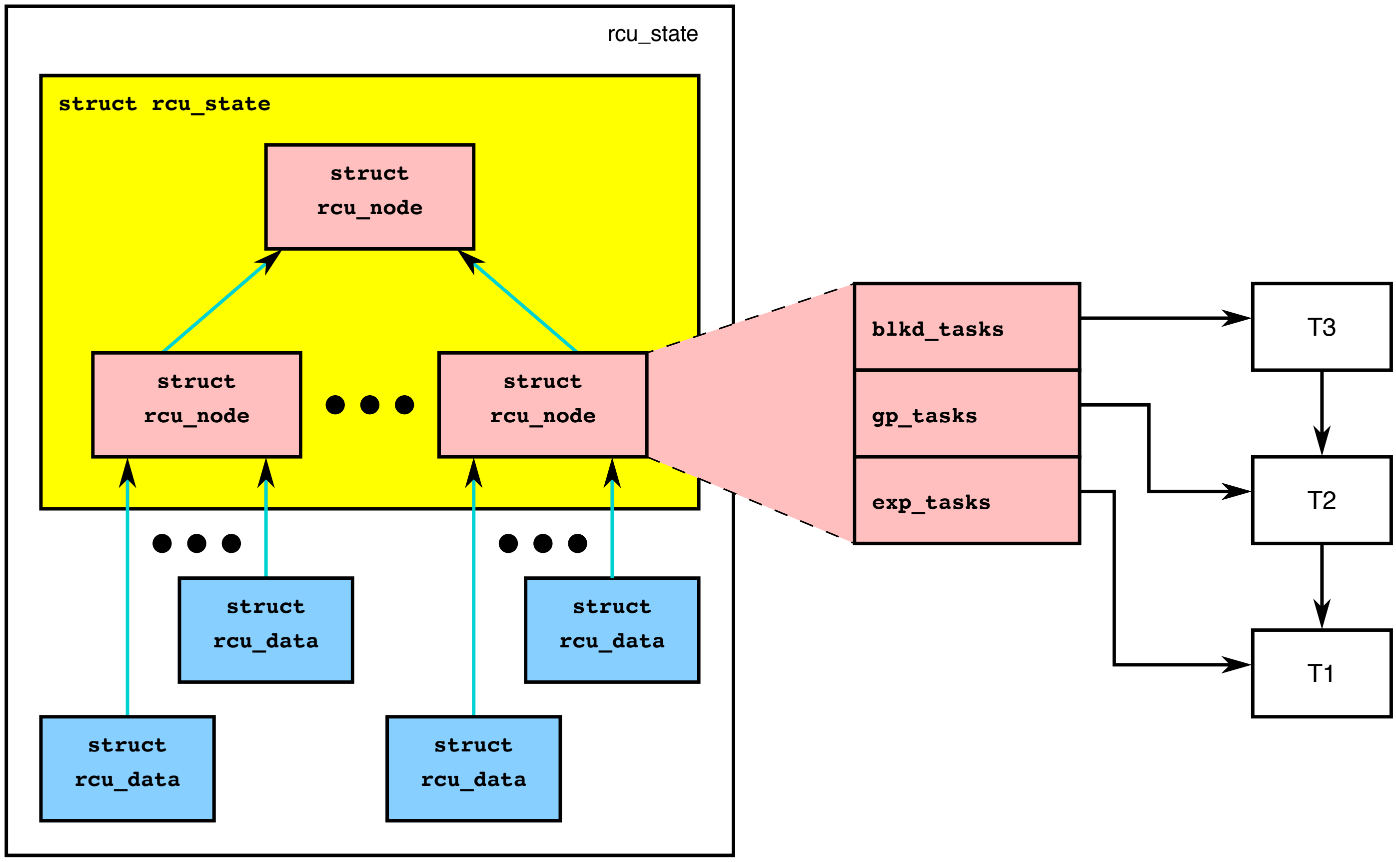
Blocked-Task Management

`PREEMPT_RCU` allows tasks to be preempted in the midst of their RCU read-side critical sections, and these tasks must be tracked explicitly. The details of exactly why and how they are tracked will be covered in a separate article on RCU read-side processing. For now, it is enough to know that the `rcu_node` structure tracks them.

```
struct list_head blkd_tasks;
struct list_head *gp_tasks;
struct list_head *exp_tasks;
bool wait_blkd_tasks;
```

The `->blkd_tasks` field is a list header for the list of blocked and preempted tasks. As tasks undergo context switches within RCU read-side critical sections, their `task_struct` structures are enqueued (via the `task_struct`'s `->rcu_node_entry` field) onto the head of the `->blkd_tasks` list for the leaf `rcu_node` structure corresponding to the CPU on which the outgoing context switch executed. As these tasks later exit their RCU read-side critical sections, they remove themselves from the list. This list is therefore in reverse time order, so that if one of the tasks is blocking the current grace period, all subsequent tasks must also be blocking that same grace period. Therefore, a single pointer into this list suffices to track all tasks blocking a given grace period. That pointer is stored in `->gp_tasks` for normal grace periods and in `->exp_tasks` for expedited grace periods. These last two fields are `NULL` if either there is no grace period in flight or if there are no blocked tasks preventing that grace period from completing. If either of these two pointers is referencing a task that removes itself from the `->blkd_tasks` list, then that task must advance the pointer to the next task on the list, or set the pointer to `NULL` if there are no subsequent tasks on the list.

For example, suppose that tasks T1, T2, and T3 are all hard-affinitied to the largest-numbered CPU in the system. Then if task T1 blocked in an RCU read-side critical section, then an expedited grace period started, then task T2 blocked in an RCU read-side critical section, then a normal grace period started, and finally task 3 blocked in an RCU read-side critical section, then the state of the last leaf `rcu_node` structure's blocked-task list would be as shown below:



Task T1 is blocking both grace periods, task T2 is blocking only the normal grace period, and task T3 is blocking neither grace period. Note that these tasks will not remove themselves from this list immediately upon resuming execution. They will instead remain on the list until they execute the outermost `rcu_read_unlock()` that ends their RCU read-side critical section.

The `->wait_blkd_tasks` field indicates whether or not the current grace period is waiting on a blocked task.

Sizing the `rcu_node` Array

The `rcu_node` array is sized via a series of C-preprocessor expressions as follows:

```
#ifdef CONFIG_RCU_FANOUT
#define RCU_FANOUT CONFIG_RCU_FANOUT
#else
# ifdef CONFIG_64BIT
# define RCU_FANOUT 64
# else
```

```
# define RCU_FANOUT 32
# endif
#endif

#ifdef CONFIG_RCU_FANOUT_LEAF
#define RCU_FANOUT_LEAF CONFIG_RCU_FANOUT_LEAF
#else
# ifdef CONFIG_64BIT
# define RCU_FANOUT_LEAF 64
# else
# define RCU_FANOUT_LEAF 32
# endif
#endif

#define RCU_FANOUT_1        (RCU_FANOUT_LEAF)
#define RCU_FANOUT_2        (RCU_FANOUT_1 * RCU_FANOUT)
#define RCU_FANOUT_3        (RCU_FANOUT_2 * RCU_FANOUT)
#define RCU_FANOUT_4        (RCU_FANOUT_3 * RCU_FANOUT)

#if NR_CPUS <= RCU_FANOUT_1
#  define RCU_NUM_LVLS       1
#  define NUM_RCU_LVL_0       1
#  define NUM_RCU_NODES       NUM_RCU_LVL_0
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0" }
#elif NR_CPUS <= RCU_FANOUT_2
#  define RCU_NUM_LVLS       2
#  define NUM_RCU_LVL_0       1
#  define NUM_RCU_LVL_1       DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
#  define NUM_RCU_NODES       (NUM_RCU_LVL_0 + NUM_RCU_LVL_1)
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1" }
#elif NR_CPUS <= RCU_FANOUT_3
#  define RCU_NUM_LVLS       3
#  define NUM_RCU_LVL_0       1
#  define NUM_RCU_LVL_1       DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_2)
#  define NUM_RCU_LVL_2       DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
#  define NUM_RCU_NODES       (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2)
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1, NUM_RCU_LVL_2 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1", "rcu_node_2" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1", "rcu_node_fqs_2" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1", "rcu_node_exp_2" }
```

```
53 #elif NR_CPUS <= RCU_FANOUT_4
54 #  define RCU_NUM_LVLS        4
55 #  define NUM_RCU_LVL_0        1
56 #  define NUM_RCU_LVL_1        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_3)
57 #  define NUM_RCU_LVL_2        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_2)
58 #  define NUM_RCU_LVL_3        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
59 #  define NUM_RCU_NODES        (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_
↪2 + NUM_RCU_LVL_3)
60 #  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1, NUM_RCU_LVL_2,
↪ NUM_RCU_LVL_3 }
61 #  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1", "rcu_node_2",
↪"rcu_node_3" }
62 #  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1", "rcu_
↪node_fqs_2", "rcu_node_fqs_3" }
63 #  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1", "rcu_
↪node_exp_2", "rcu_node_exp_3" }
64 #else
65 # error "CONFIG_RCU_FANOUT insufficient for NR_CPUS"
66 #endif
```

The maximum number of levels in the `rcu_node` structure is currently limited to four, as specified by lines 21-24 and the structure of the subsequent "if" statement. For 32-bit systems, this allows 16*32*32*32=524,288 CPUs, which should be sufficient for the next few years at least. For 64-bit systems, 16*64*64*64=4,194,304 CPUs is allowed, which should see us through the next decade or so. This four-level tree also allows kernels built with `CONFIG_RCU_FANOUT=8` to support up to 4096 CPUs, which might be useful in very large systems having eight CPUs per socket (but please note that no one has yet shown any measurable performance degradation due to misaligned socket and `rcu_node` boundaries). In addition, building kernels with a full four levels of `rcu_node` tree permits better testing of RCU's combining-tree code.

The `RCU_FANOUT` symbol controls how many children are permitted at each non-leaf level of the `rcu_node` tree. If the `CONFIG_RCU_FANOUT` Kconfig option is not specified, it is set based on the word size of the system, which is also the Kconfig default.

The `RCU_FANOUT_LEAF` symbol controls how many CPUs are handled by each leaf `rcu_node` structure. Experience has shown that allowing a given leaf `rcu_node` structure to handle 64 CPUs, as permitted by the number of bits in the `->qsmask` field on a 64-bit system, results in excessive contention for the leaf `rcu_node` structures' `->lock` fields. The number of CPUs per leaf `rcu_node` structure is therefore limited to 16 given the default value of `CONFIG_RCU_FANOUT_LEAF`. If `CONFIG_RCU_FANOUT_LEAF` is unspecified, the value selected is based on the word size of the system, just as for `CONFIG_RCU_FANOUT`. Lines 11-19 perform this computation.

Lines 21-24 compute the maximum number of CPUs supported by a single-level (which contains a single `rcu_node` structure), two-level, three-level, and four-level `rcu_node` tree, respectively, given the fanout specified by `RCU_FANOUT` and `RCU_FANOUT_LEAF`. These numbers of CPUs are retained in the `RCU_FANOUT_1`, `RCU_FANOUT_2`, `RCU_FANOUT_3`, and `RCU_FANOUT_4` C-preprocessor variables, respectively.

These variables are used to control the C-preprocessor `#if` statement spanning lines 26-66 that computes the number of `rcu_node` structures required for each level of the tree, as well as the number of levels required. The number of levels is placed in the `NUM_RCU_LVLS` C-preprocessor variable by lines 27, 35, 44, and 54. The number of `rcu_node` structures for the topmost level of the tree is always exactly one, and this value is unconditionally placed into NUM_RCU_LVL_0 by lines 28, 36, 45, and 55. The rest of the levels (if any) of the rcu_node tree are computed by dividing the maximum number of CPUs by the fanout supported by the number of levels from the current level down, rounding up. This computation is performed by lines 37, 46-47, and 56-58. Lines 31-33, 40-42, 50-52, and 62-63 create initializers for lockdep lock-class names. Finally, lines 64-66 produce an error if the maximum number of CPUs is too large for the specified fanout.

The `rcu_segcblist` Structure

The `rcu_segcblist` structure maintains a segmented list of callbacks as follows:

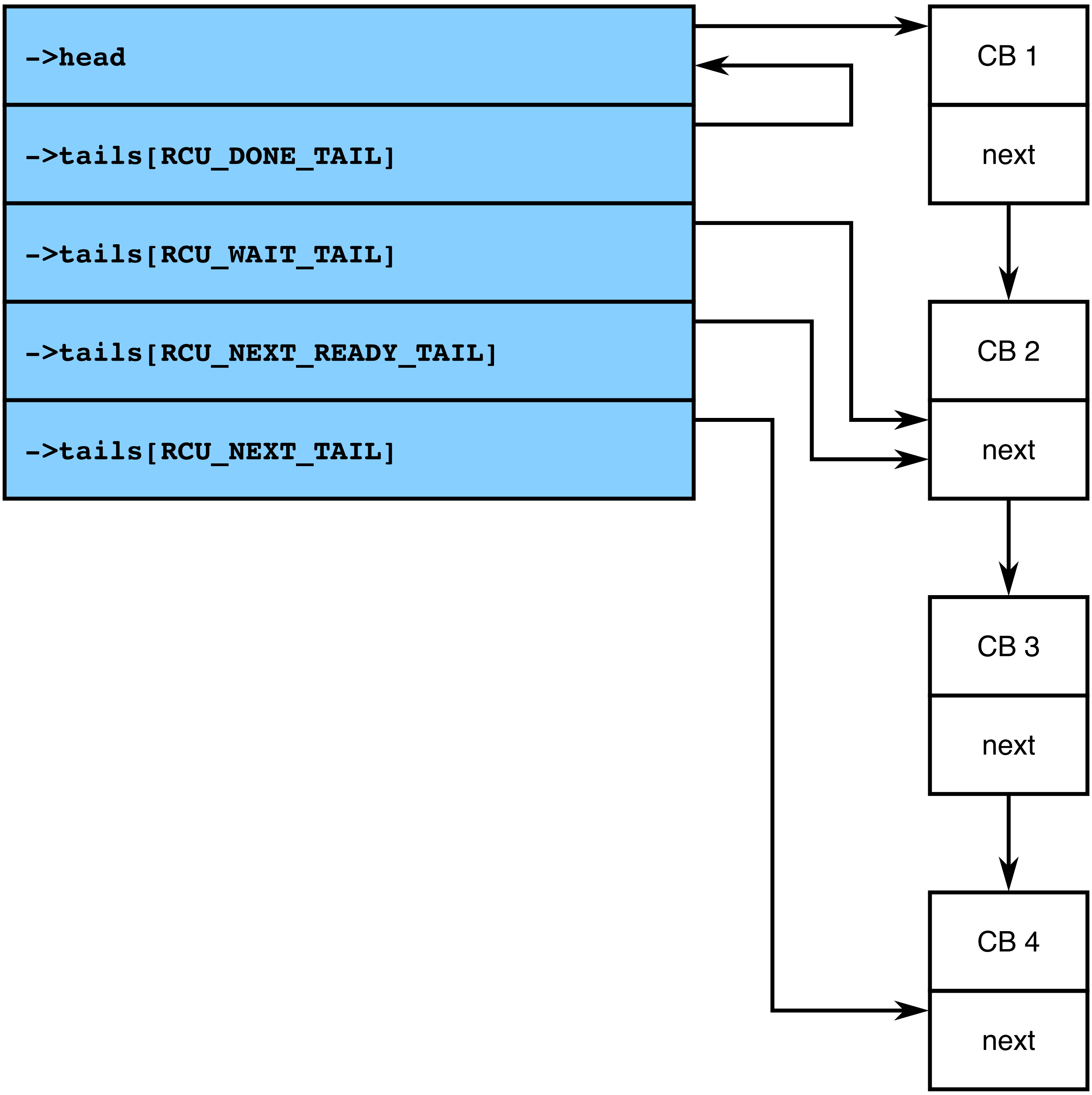
```
#define RCU_DONE_TAIL        0
#define RCU_WAIT_TAIL        1
#define RCU_NEXT_READY_TAIL  2
#define RCU_NEXT_TAIL        3
#define RCU_CBLIST_NSEGS     4

struct rcu_segcblist {
  struct rcu_head *head;
  struct rcu_head **tails[RCU_CBLIST_NSEGS];
  unsigned long gp_seq[RCU_CBLIST_NSEGS];
  long len;
  long len_lazy;
};
```

The segments are as follows:

1. `RCU_DONE_TAIL`: Callbacks whose grace periods have elapsed. These callbacks are ready to be invoked.
2. `RCU_WAIT_TAIL`: Callbacks that are waiting for the current grace period. Note that different CPUs can have different ideas about which grace period is current, hence the `->gp_seq` field.
3. `RCU_NEXT_READY_TAIL`: Callbacks waiting for the next grace period to start.
4. `RCU_NEXT_TAIL`: Callbacks that have not yet been associated with a grace period.

The `->head` pointer references the first callback or is `NULL` if the list contains no callbacks (which is *not* the same as being empty). Each element of the `->tails[]` array references the `->next` pointer of the last callback in the corresponding segment of the list, or the list's `->head` pointer if that segment and all previous segments are empty. If the corresponding segment is empty but some previous segment is not empty, then the array element is identical to its predecessor. Older callbacks are closer to the head of the list, and new callbacks are added at the tail. This relationship between the `->head` pointer, the `->tails[]` array, and the callbacks is shown in this diagram:

In this figure, the `->head` pointer references the first RCU callback in the list. The `->tails[RCU_DONE_TAIL]` array element references the `->head` pointer itself, indicating that none of the callbacks is ready to invoke. The `->tails[RCU_WAIT_TAIL]` array element references callback CB 2's `->next` pointer, which indicates that CB 1 and CB 2 are both waiting on the current grace period, give or take possible disagreements about exactly which grace period is the current one. The `->tails[RCU_NEXT_READY_TAIL]` array element references the


->head
->tails[RCU_DONE_TAIL]
->tails[RCU_WAIT_TAIL]
->tails[RCU_NEXT_READY_TAIL]
->tails[RCU_NEXT_TAIL]
CB 1
next
CB 2
next
CB 3
next
CB 4
next

same RCU callback that `->tails[RCU_WAIT_TAIL]` does, which indicates that there are no callbacks waiting on the next RCU grace period. The `->tails[RCU_NEXT_TAIL]` array element references CB 4's `->next` pointer, indicating that all the remaining RCU callbacks have not yet been assigned to an RCU grace period. Note that the `->tails[RCU_NEXT_TAIL]` array element always references the last RCU callback's `->next` pointer unless the callback list is empty, in which case it references the `->head` pointer.

There is one additional important special case for the `->tails[RCU_NEXT_TAIL]` array element: It can be `NULL` when this list is *disabled*. Lists are disabled when the corresponding CPU is offline or when the corresponding CPU's callbacks are offloaded to a kthread, both of which are described elsewhere.

CPUs advance their callbacks from the `RCU_NEXT_TAIL` to the `RCU_NEXT_READY_TAIL` to the `RCU_WAIT_TAIL` to the `RCU_DONE_TAIL` list segments as grace periods advance.

The `->gp_seq[]` array records grace-period numbers corresponding to the list segments. This is what allows different CPUs to have different ideas as to which is the current grace period while still avoiding premature invocation of their callbacks. In particular, this allows CPUs that go idle for extended periods to determine which of their callbacks are ready to be invoked after reawakening.

The `->len` counter contains the number of callbacks in `->head`, and the `->len_lazy` contains the number of those callbacks that are known to only free memory, and whose invocation can therefore be safely deferred.

Important: It is the `->len` field that determines whether or not there are callbacks associated with this `rcu_segcblist` structure, *not* the `->head` pointer. The reason for this is that all the ready-to-invoke callbacks (that is, those in the `RCU_DONE_TAIL` segment) are extracted all at once at callback-invocation time (`rcu_do_batch`), due to which `->head` may be set to NULL if there are no not-done callbacks remaining in the `rcu_segcblist`. If callback invocation must be postponed, for example, because a high-priority process just woke up on this CPU, then the remaining callbacks are placed back on the `RCU_DONE_TAIL` segment and `->head` once again points to the start of the segment. In short, the head field can briefly be `NULL` even though the CPU has callbacks present the entire time. Therefore, it is not appropriate to test the `->head` pointer for `NULL`.

In contrast, the `->len` and `->len_lazy` counts are adjusted only after the corresponding callbacks have been invoked. This means that the `->len` count is zero only if the `rcu_segcblist` structure really is devoid of callbacks. Of course, off-CPU sampling of the `->len` count requires careful use of appropriate synchronization, for example, memory barriers. This synchronization can be a bit subtle, particularly in the case of `rcu_barrier()`.

The `rcu_data` Structure

The `rcu_data` maintains the per-CPU state for the RCU subsystem. The fields in this structure may be accessed only from the corresponding CPU (and from tracing) unless otherwise stated. This structure is the focus of quiescent-state detection and RCU callback queuing. It also tracks its relationship to the corresponding leaf `rcu_node` structure to allow more-efficient propagation of quiescent states up the `rcu_node` combining tree. Like the `rcu_node` structure, it provides a local copy of the grace-period information to allow for-free synchronized access to this information from the corresponding CPU. Finally, this structure records past dyntick-idle state for the corresponding CPU and also tracks statistics.

The rcu_data structure's fields are discussed, singly and in groups, in the following sections.

Connection to Other Data Structures

This portion of the rcu_data structure is declared as follows:

```
int cpu;
struct rcu_node *mynode;
unsigned long grpmask;
bool beenonline;
```

The ->cpu field contains the number of the corresponding CPU and the ->mynode field references the corresponding rcu_node structure. The ->mynode is used to propagate quiescent states up the combining tree. These two fields are constant and therefore do not require synchronization.

The ->grpmask field indicates the bit in the ->mynode->qsmask corresponding to this rcu_data structure, and is also used when propagating quiescent states. The ->beenonline flag is set whenever the corresponding CPU comes online, which means that the debugfs tracing need not dump out any rcu_data structure for which this flag is not set.

Quiescent-State and Grace-Period Tracking

This portion of the rcu_data structure is declared as follows:

```
unsigned long gp_seq;
unsigned long gp_seq_needed;
bool cpu_no_qs;
bool core_needs_qs;
bool gpwrap;
```

The ->gp_seq field is the counterpart of the field of the same name in the rcu_state and rcu_node structures. The ->gp_seq_needed field is the counterpart of the field of the same name in the rcu_node structure. They may each lag up to one behind their rcu_node counterparts, but in CONFIG_NO_HZ_IDLE and CONFIG_NO_HZ_FULL kernels can lag arbitrarily far behind for CPUs in dyntick-idle mode (but these counters will catch up upon exit from dyntick-idle mode). If the lower two bits of a given rcu_data structure's ->gp_seq are zero, then this rcu_data structure believes that RCU is idle.

Quick Quiz:

All this replication of the grace period numbers can only cause massive confusion. Why not just keep a global sequence number and be done with it???

Answer:

Because if there was only a single global sequence numbers, there would need to be a single global lock to allow safely accessing and updating it. And if we are not going to have a single global lock, we need to carefully manage the numbers on a per-node basis. Recall from the answer to a previous Quick Quiz that the consequences of applying a previously sampled quiescent state to the wrong grace period are quite severe.

The `->cpu_no_qs` flag indicates that the CPU has not yet passed through a quiescent state, while the `->core_needs_qs` flag indicates that the RCU core needs a quiescent state from the corresponding CPU. The `->gpwrap` field indicates that the corresponding CPU has remained idle for so long that the `gp_seq` counter is in danger of overflow, which will cause the CPU to disregard the values of its counters on its next exit from idle.

RCU Callback Handling

In the absence of CPU-hotplug events, RCU callbacks are invoked by the same CPU that registered them. This is strictly a cache-locality optimization: callbacks can and do get invoked on CPUs other than the one that registered them. After all, if the CPU that registered a given callback has gone offline before the callback can be invoked, there really is no other choice.

This portion of the `rcu_data` structure is declared as follows:

```
struct rcu_segcblist cblist;
long qlen_last_fqs_check;
unsigned long n_cbs_invoked;
unsigned long n_nocbs_invoked;
unsigned long n_cbs_orphaned;
unsigned long n_cbs_adopted;
unsigned long n_force_qs_snap;
long blimit;
```

The `->cblist` structure is the segmented callback list described earlier. The CPU advances the callbacks in its `rcu_data` structure whenever it notices that another RCU grace period has completed. The CPU detects the completion of an RCU grace period by noticing that the value of its `rcu_data` structure's `->gp_seq` field differs from that of its leaf `rcu_node` structure. Recall that each `rcu_node` structure's `->gp_seq` field is updated at the beginnings and ends of each grace period.

The `->qlen_last_fqs_check` and `->n_force_qs_snap` coordinate the forcing of quiescent states from `call_rcu()` and friends when callback lists grow excessively long.

The `->n_cbs_invoked`, `->n_cbs_orphaned`, and `->n_cbs_adopted` fields count the number of callbacks invoked, sent to other CPUs when this CPU goes offline, and received from other CPUs when those other CPUs go offline. The `->n_nocbs_invoked` is used when the CPU's callbacks are offloaded to a kthread.

Finally, the `->blimit` counter is the maximum number of RCU callbacks that may be invoked at a given time.

Dyntick-Idle Handling

This portion of the rcu_data structure is declared as follows:

```
int dynticks_snap;
unsigned long dynticks_fqs;
```

The ->dynticks_snap field is used to take a snapshot of the corresponding CPU's dyntick-idle state when forcing quiescent states, and is therefore accessed from other CPUs. Finally, the ->dynticks_fqs field is used to count the number of times this CPU is determined to be in dyntick-idle state, and is used for tracing and debugging purposes.

This portion of the rcu_data structure is declared as follows:

```
long dynticks_nesting;
long dynticks_nmi_nesting;
atomic_t dynticks;
bool rcu_need_heavy_qs;
bool rcu_urgent_qs;
```

These fields in the rcu_data structure maintain the per-CPU dyntick-idle state for the corresponding CPU. The fields may be accessed only from the corresponding CPU (and from tracing) unless otherwise stated.

The ->dynticks_nesting field counts the nesting depth of process execution, so that in normal circumstances this counter has value zero or one. NMIs, irqs, and tracers are counted by the ->dynticks_nmi_nesting field. Because NMIs cannot be masked, changes to this variable have to be undertaken carefully using an algorithm provided by Andy Lutomirski. The initial transition from idle adds one, and nested transitions add two, so that a nesting level of five is represented by a ->dynticks_nmi_nesting value of nine. This counter can therefore be thought of as counting the number of reasons why this CPU cannot be permitted to enter dyntick-idle mode, aside from process-level transitions.

However, it turns out that when running in non-idle kernel context, the Linux kernel is fully capable of entering interrupt handlers that never exit and perhaps also vice versa. Therefore, whenever the ->dynticks_nesting field is incremented up from zero, the ->dynticks_nmi_nesting field is set to a large positive number, and whenever the ->dynticks_nesting field is decremented down to zero, the ->dynticks_nmi_nesting field is set to zero. Assuming that the number of misnested interrupts is not sufficient to overflow the counter, this approach corrects the ->dynticks_nmi_nesting field every time the corresponding CPU enters the idle loop from process context.

The ->dynticks field counts the corresponding CPU's transitions to and from either dyntick-idle or user mode, so that this counter has an even value when the CPU is in dyntick-idle mode or user mode and an odd value otherwise. The transitions to/from user mode need to be counted for user mode adaptive-ticks support (see Documentation/timers/no_hz.rst).

The ->rcu_need_heavy_qs field is used to record the fact that the RCU core code would really like to see a quiescent state from the corresponding CPU, so much so that it is willing to call for heavy-weight dyntick-counter operations. This flag is checked by RCU's context-switch and cond_resched() code, which provide a momentary idle sojourn in response.

Finally, the ->rcu_urgent_qs field is used to record the fact that the RCU core code would really like to see a quiescent state from the corresponding CPU, with the various other fields indicating just how badly RCU wants this quiescent state. This flag is checked by RCU's context-switch path (rcu_note_context_switch) and the cond_resched code.

Quick Quiz:

Why not simply combine the ->dynticks_nesting and ->dynticks_nmi_nesting counters into a single counter that just counts the number of reasons that the corresponding CPU is non-idle?

Answer:

Because this would fail in the presence of interrupts whose handlers never return and of handlers that manage to return from a made-up interrupt.

Additional fields are present for some special-purpose builds, and are discussed separately.

The rcu_head Structure

Each rcu_head structure represents an RCU callback. These structures are normally embedded within RCU-protected data structures whose algorithms use asynchronous grace periods. In contrast, when using algorithms that block waiting for RCU grace periods, RCU users need not provide rcu_head structures.

The rcu_head structure has fields as follows:

```
struct rcu_head *next;
void (*func)(struct rcu_head *head);
```

The ->next field is used to link the rcu_head structures together in the lists within the rcu_data structures. The ->func field is a pointer to the function to be called when the callback is ready to be invoked, and this function is passed a pointer to the rcu_head structure. However, kfree_rcu() uses the ->func field to record the offset of the rcu_head structure within the enclosing RCU-protected data structure.

Both of these fields are used internally by RCU. From the viewpoint of RCU users, this structure is an opaque "cookie".

Quick Quiz:

Given that the callback function ->func is passed a pointer to the rcu_head structure, how is that function supposed to find the beginning of the enclosing RCU-protected data structure?

Answer:

In actual practice, there is a separate callback function per type of RCU-protected data structure. The callback function can therefore use the container_of() macro in the Linux kernel (or other pointer-manipulation facilities in other software environments) to find the beginning of the enclosing structure.

RCU-Specific Fields in the task_struct Structure

The CONFIG_PREEMPT_RCU implementation uses some additional fields in the task_struct structure:

```
#ifdef CONFIG_PREEMPT_RCU
  int rcu_read_lock_nesting;
  union rcu_special rcu_read_unlock_special;
  struct list_head rcu_node_entry;
  struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
  unsigned long rcu_tasks_nvcsw;
  bool rcu_tasks_holdout;
  struct list_head rcu_tasks_holdout_list;
  int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */
```

The ->rcu_read_lock_nesting field records the nesting level for RCU read-side critical sections, and the ->rcu_read_unlock_special field is a bitmask that records special conditions that require rcu_read_unlock() to do additional work. The ->rcu_node_entry field is used to form lists of tasks that have blocked within preemptible-RCU read-side critical sections and the ->rcu_blocked_node field references the rcu_node structure whose list this task is a member of, or NULL if it is not blocked within a preemptible-RCU read-side critical section.

The ->rcu_tasks_nvcsw field tracks the number of voluntary context switches that this task had undergone at the beginning of the current tasks-RCU grace period, ->rcu_tasks_holdout is set if the current tasks-RCU grace period is waiting on this task, ->rcu_tasks_holdout_list is a list element enqueuing this task on the holdout list, and ->rcu_tasks_idle_cpu tracks which CPU this idle task is running, but only if the task is currently running, that is, if the CPU is currently idle.

Accessor Functions

The following listing shows the rcu_get_root(), rcu_for_each_node_breadth_first and rcu_for_each_leaf_node() function and macros:

```
static struct rcu_node *rcu_get_root(struct rcu_state *rsp)
{
  return &rsp->node[0];
}

#define rcu_for_each_node_breadth_first(rsp, rnp) \
  for ((rnp) = &(rsp)->node[0]; \
       (rnp) < &(rsp)->node[NUM_RCU_NODES]; (rnp)++)

#define rcu_for_each_leaf_node(rsp, rnp) \
  for ((rnp) = (rsp)->level[NUM_RCU_LVLS - 1]; \
       (rnp) < &(rsp)->node[NUM_RCU_NODES]; (rnp)++)
```

The rcu_get_root() simply returns a pointer to the first element of the specified rcu_state structure's ->node[] array, which is the root rcu_node structure.

As noted earlier, the rcu_for_each_node_breadth_first() macro takes advantage of the layout of the rcu_node structures in the rcu_state structure's ->node[] array, performing a breadth-first traversal by simply traversing the array in order. Similarly, the rcu_for_each_leaf_node() macro traverses only the last part of the array, thus traversing only the leaf rcu_node structures.

Quick Quiz:
What does rcu_for_each_leaf_node() do if the rcu_node tree contains only a single node?
Answer:
In the single-node case, rcu_for_each_leaf_node() traverses the single node.

Summary

So the state of RCU is represented by an rcu_state structure, which contains a combining tree of rcu_node and rcu_data structures. Finally, in CONFIG_NO_HZ_IDLE kernels, each CPU's dyntick-idle state is tracked by dynticks-related fields in the rcu_data structure. If you made it this far, you are well prepared to read the code walkthroughs in the other articles in this series.

Acknowledgments

I owe thanks to Cyrill Gorcunov, Mathieu Desnoyers, Dhaval Giani, Paul Turner, Abhishek Srivastava, Matt Kowalczyk, and Serge Hallyn for helping me get this document into a more human-readable state.

Legal Statement

4.6 Linux kernel memory barriers

```
                         ============================
                         LINUX KERNEL MEMORY BARRIERS
                         ============================

By: David Howells <dhowells@redhat.com>
    Paul E. McKenney <paulmck@linux.ibm.com>
    Will Deacon <will.deacon@arm.com>
    Peter Zijlstra <peterz@infradead.org>

==========
DISCLAIMER
==========

This document is not a specification; it is intentionally (for the sake of
brevity) and unintentionally (due to being human) incomplete. This document is
```

meant as a guide to using the various memory barriers provided by Linux, but in case of any doubt (and there are many) please ask. Some doubts may be resolved by referring to the formal memory consistency model and related documentation at tools/memory-model/. Nevertheless, even this memory model should be viewed as the collective opinion of its maintainers rather than as an infallible oracle.

To repeat, this document is not a specification of what Linux expects from hardware.

The purpose of this document is twofold:

(1) to specify the minimum functionality that one can rely on for any particular barrier, and

(2) to provide a guide as to how to use the barriers that are available.

Note that an architecture can provide more than the minimum requirement for any particular barrier, but if the architecture provides less than that, that architecture is incorrect.

Note also that it is possible that a barrier may be a no-op for an architecture because the way that arch works renders an explicit barrier unnecessary in that case.

CONTENTS

```
     - Interprocessor interaction.
     - Atomic operations.
     - Accessing devices.
     - Interrupts.

 (*) Kernel I/O barrier effects.

 (*) Assumed minimum execution ordering model.

 (*) The effects of the cpu cache.

     - Cache coherency.
     - Cache coherency vs DMA.
     - Cache coherency vs MMIO.

 (*) The things CPUs get up to.

     - And then there's the Alpha.
     - Virtual Machine Guests.

 (*) Example uses.

     - Circular buffers.

 (*) References.


============================
ABSTRACT MEMORY ACCESS MODEL
============================

Consider the following abstract model of the system:

                            :                :
                            :                :
                            :                :
                +-------+   :   +--------+   :   +-------+
                |       |   :   |        |   :   |       |
                |       |   :   |        |   :   |       |
                | CPU 1 |<----->| Memory |<----->| CPU 2 |
                |       |   :   |        |   :   |       |
                |       |   :   |        |   :   |       |
                +-------+   :   +--------+   :   +-------+
                    ^       :       ^        :       ^
                    |       :       |        :       |
                    |       :       |        :       |
                    |       :       v        :       |
                    |       :   +--------+   :       |
                    |       :   |        |   :       |
                    |       :   |        |   :       |
                    +---------->| Device |<----------+
                            :   |        |   :
                            :   |        |   :
                            :   +--------+   :
                            :                :

Each CPU executes a program that generates memory access operations.  In the
abstract CPU, memory operation ordering is very relaxed, and a CPU may actually
perform the memory operations in any order it likes, provided program causality
appears to be maintained.  Similarly, the compiler may also arrange the
instructions it emits in any order it likes, provided it doesn't affect the
apparent operation of the program.
```

```
So in the above diagram, the effects of the memory operations performed by a
CPU are perceived by the rest of the system as the operations cross the
interface between the CPU and rest of the system (the dotted lines).


For example, consider the following sequence of events:

        CPU 1           CPU 2
        =============== ===============
        { A == 1; B == 2 }
        A = 3;          x = B;
        B = 4;          y = A;

The set of accesses as seen by the memory system in the middle can be arranged
in 24 different combinations:

        STORE A=3,      STORE B=4,      y=LOAD A->3,    x=LOAD B->4
        STORE A=3,      STORE B=4,      x=LOAD B->4,    y=LOAD A->3
        STORE A=3,      y=LOAD A->3,    STORE B=4,      x=LOAD B->4
        STORE A=3,      y=LOAD A->3,    x=LOAD B->2,    STORE B=4
        STORE A=3,      x=LOAD B->2,    STORE B=4,      y=LOAD A->3
        STORE A=3,      x=LOAD B->2,    y=LOAD A->3,    STORE B=4
        STORE B=4,      STORE A=3,      y=LOAD A->3,    x=LOAD B->4
        STORE B=4, ...
        ...

and can thus result in four different combinations of values:

        x == 2, y == 1
        x == 2, y == 3
        x == 4, y == 1
        x == 4, y == 3


Furthermore, the stores committed by a CPU to the memory system may not be
perceived by the loads made by another CPU in the same order as the stores were
committed.


As a further example, consider this sequence of events:

        CPU 1           CPU 2
        =============== ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;          Q = P;
        P = &B;         D = *Q;

There is an obvious address dependency here, as the value loaded into D depends
on the address retrieved from P by CPU 2.  At the end of the sequence, any of
the following results are possible:

        (Q == &A) and (D == 1)
        (Q == &B) and (D == 2)
        (Q == &B) and (D == 4)

Note that CPU 2 will never try and load C into D because the CPU will load P
into Q before issuing the load of *Q.


DEVICE OPERATIONS
-----------------
```

```
Some devices present their control interfaces as collections of memory
locations, but the order in which the control registers are accessed is very
important.  For instance, imagine an ethernet card with a set of internal
registers that are accessed through an address port register (A) and a data
port register (D).  To read internal register 5, the following code might then
be used:

        *A = 5;
        x = *D;

but this might show up as either of the following two sequences:

        STORE *A = 5, x = LOAD *D
        x = LOAD *D, STORE *A = 5

the second of which will almost certainly result in a malfunction, since it set
the address _after_ attempting to read the register.


GUARANTEES
----------

There are some minimal guarantees that may be expected of a CPU:

 (*) On any given CPU, dependent memory accesses will be issued in order, with
     respect to itself.  This means that for:

        Q = READ_ONCE(P); D = READ_ONCE(*Q);

     the CPU will issue the following memory operations:

        Q = LOAD P, D = LOAD *Q

     and always in that order.  However, on DEC Alpha, READ_ONCE() also
     emits a memory-barrier instruction, so that a DEC Alpha CPU will
     instead issue the following memory operations:

        Q = LOAD P, MEMORY_BARRIER, D = LOAD *Q, MEMORY_BARRIER

     Whether on DEC Alpha or not, the READ_ONCE() also prevents compiler
     mischief.

 (*) Overlapping loads and stores within a particular CPU will appear to be
     ordered within that CPU.  This means that for:

        a = READ_ONCE(*X); WRITE_ONCE(*X, b);

     the CPU will only issue the following sequence of memory operations:

        a = LOAD *X, STORE *X = b

     And for:

        WRITE_ONCE(*X, c); d = READ_ONCE(*X);

     the CPU will only issue:

        STORE *X = c, d = LOAD *X

     (Loads and stores overlap if they are targeted at overlapping pieces of
     memory).

And there are a number of things that _must_ or _must_not_ be assumed:
```

```
 (*) It _must_not_ be assumed that the compiler will do what you want
     with memory references that are not protected by READ_ONCE() and
     WRITE_ONCE().  Without them, the compiler is within its rights to
     do all sorts of "creative" transformations, which are covered in
     the COMPILER BARRIER section.

 (*) It _must_not_ be assumed that independent loads and stores will be issued
     in the order given.  This means that for:

        X = *A; Y = *B; *D = Z;

     we may get any of the following sequences:

        X = LOAD *A,  Y = LOAD *B,  STORE *D = Z
        X = LOAD *A,  STORE *D = Z, Y = LOAD *B
        Y = LOAD *B,  X = LOAD *A,  STORE *D = Z
        Y = LOAD *B,  STORE *D = Z, X = LOAD *A
        STORE *D = Z, X = LOAD *A,  Y = LOAD *B
        STORE *D = Z, Y = LOAD *B,  X = LOAD *A

 (*) It _must_ be assumed that overlapping memory accesses may be merged or
     discarded.  This means that for:

        X = *A; Y = *(A + 4);

     we may get any one of the following sequences:

        X = LOAD *A; Y = LOAD *(A + 4);
        Y = LOAD *(A + 4); X = LOAD *A;
        {X, Y} = LOAD {*A, *(A + 4) };

     And for:

        *A = X; *(A + 4) = Y;

     we may get any of:

        STORE *A = X; STORE *(A + 4) = Y;
        STORE *(A + 4) = Y; STORE *A = X;
        STORE {*A, *(A + 4) } = {X, Y};

And there are anti-guarantees:

 (*) These guarantees do not apply to bitfields, because compilers often
     generate code to modify these using non-atomic read-modify-write
     sequences.  Do not attempt to use bitfields to synchronize parallel
     algorithms.

 (*) Even in cases where bitfields are protected by locks, all fields
     in a given bitfield must be protected by one lock.  If two fields
     in a given bitfield are protected by different locks, the compiler's
     non-atomic read-modify-write sequences can cause an update to one
     field to corrupt the value of an adjacent field.

 (*) These guarantees apply only to properly aligned and sized scalar
     variables.  "Properly sized" currently means variables that are
     the same size as "char", "short", "int" and "long".  "Properly
     aligned" means the natural alignment, thus no constraints for
     "char", two-byte alignment for "short", four-byte alignment for
     "int", and either four-byte or eight-byte alignment for "long",
     on 32-bit and 64-bit systems, respectively.  Note that these
     guarantees were introduced into the C11 standard, so beware when
```

```
     using older pre-C11 compilers (for example, gcc 4.6).  The portion
     of the standard containing this guarantee is Section 3.14, which
     defines "memory location" as follows:

        memory location
                either an object of scalar type, or a maximal sequence
                of adjacent bit-fields all having nonzero width

                NOTE 1: Two threads of execution can update and access
                separate memory locations without interfering with
                each other.

                NOTE 2: A bit-field and an adjacent non-bit-field member
                are in separate memory locations. The same applies
                to two bit-fields, if one is declared inside a nested
                structure declaration and the other is not, or if the two
                are separated by a zero-length bit-field declaration,
                or if they are separated by a non-bit-field member
                declaration. It is not safe to concurrently update two
                bit-fields in the same structure if all members declared
                between them are also bit-fields, no matter what the
                sizes of those intervening bit-fields happen to be.


=========================
WHAT ARE MEMORY BARRIERS?
=========================

As can be seen above, independent memory operations are effectively performed
in random order, but this can be a problem for CPU-CPU interaction and for I/O.
What is required is some way of intervening to instruct the compiler and the
CPU to restrict the order.

Memory barriers are such interventions.  They impose a perceived partial
ordering over the memory operations on either side of the barrier.

Such enforcement is important because the CPUs and other devices in a system
can use a variety of tricks to improve performance, including reordering,
deferral and combination of memory operations; speculative loads; speculative
branch prediction and various types of caching.  Memory barriers are used to
override or suppress these tricks, allowing the code to sanely control the
interaction of multiple CPUs and/or devices.


VARIETIES OF MEMORY BARRIER
---------------------------

Memory barriers come in four basic varieties:

 (1) Write (or store) memory barriers.

     A write memory barrier gives a guarantee that all the STORE operations
     specified before the barrier will appear to happen before all the STORE
     operations specified after the barrier with respect to the other
     components of the system.

     A write barrier is a partial ordering on stores only; it is not required
     to have any effect on loads.

     A CPU can be viewed as committing a sequence of store operations to the
     memory system as time progresses.  All stores _before_ a write barrier
     will occur _before_ all the stores after the write barrier.
```

```
    [!] Note that write barriers should normally be paired with read or
    address-dependency barriers; see the "SMP barrier pairing" subsection.


(2) Address-dependency barriers (historical).

    An address-dependency barrier is a weaker form of read barrier.  In the
    case where two loads are performed such that the second depends on the
    result of the first (eg: the first load retrieves the address to which
    the second load will be directed), an address-dependency barrier would
    be required to make sure that the target of the second load is updated
    after the address obtained by the first load is accessed.

    An address-dependency barrier is a partial ordering on interdependent
    loads only; it is not required to have any effect on stores, independent
    loads or overlapping loads.

    As mentioned in (1), the other CPUs in the system can be viewed as
    committing sequences of stores to the memory system that the CPU being
    considered can then perceive.  An address-dependency barrier issued by
    the CPU under consideration guarantees that for any load preceding it,
    if that load touches one of a sequence of stores from another CPU, then
    by the time the barrier completes, the effects of all the stores prior to
    that touched by the load will be perceptible to any loads issued after
    the address-dependency barrier.

    See the "Examples of memory barrier sequences" subsection for diagrams
    showing the ordering constraints.

    [!] Note that the first load really has to have an _address_ dependency and
    not a control dependency.  If the address for the second load is dependent
    on the first load, but the dependency is through a conditional rather than
    actually loading the address itself, then it's a _control_ dependency and
    a full read barrier or better is required.  See the "Control dependencies"
    subsection for more information.

    [!] Note that address-dependency barriers should normally be paired with
    write barriers; see the "SMP barrier pairing" subsection.

    [!] Kernel release v5.9 removed kernel APIs for explicit address-
    dependency barriers.  Nowadays, APIs for marking loads from shared
    variables such as READ_ONCE() and rcu_dereference() provide implicit
    address-dependency barriers.

(3) Read (or load) memory barriers.

    A read barrier is an address-dependency barrier plus a guarantee that all
    the LOAD operations specified before the barrier will appear to happen
    before all the LOAD operations specified after the barrier with respect to
    the other components of the system.

    A read barrier is a partial ordering on loads only; it is not required to
    have any effect on stores.

    Read memory barriers imply address-dependency barriers, and so can
    substitute for them.

    [!] Note that read barriers should normally be paired with write barriers;
    see the "SMP barrier pairing" subsection.


(4) General memory barriers.
```

```
     A general memory barrier gives a guarantee that all the LOAD and STORE
     operations specified before the barrier will appear to happen before all
     the LOAD and STORE operations specified after the barrier with respect to
     the other components of the system.

     A general memory barrier is a partial ordering over both loads and stores.

     General memory barriers imply both read and write memory barriers, and so
     can substitute for either.


And a couple of implicit varieties:

 (5) ACQUIRE operations.

     This acts as a one-way permeable barrier.  It guarantees that all memory
     operations after the ACQUIRE operation will appear to happen after the
     ACQUIRE operation with respect to the other components of the system.
     ACQUIRE operations include LOCK operations and both smp_load_acquire()
     and smp_cond_load_acquire() operations.

     Memory operations that occur before an ACQUIRE operation may appear to
     happen after it completes.

     An ACQUIRE operation should almost always be paired with a RELEASE
     operation.


 (6) RELEASE operations.

     This also acts as a one-way permeable barrier.  It guarantees that all
     memory operations before the RELEASE operation will appear to happen
     before the RELEASE operation with respect to the other components of the
     system. RELEASE operations include UNLOCK operations and
     smp_store_release() operations.

     Memory operations that occur after a RELEASE operation may appear to
     happen before it completes.

     The use of ACQUIRE and RELEASE operations generally precludes the need
     for other sorts of memory barrier.  In addition, a RELEASE+ACQUIRE pair is
     -not- guaranteed to act as a full memory barrier.  However, after an
     ACQUIRE on a given variable, all memory accesses preceding any prior
     RELEASE on that same variable are guaranteed to be visible.  In other
     words, within a given variable's critical section, all accesses of all
     previous critical sections for that variable are guaranteed to have
     completed.

     This means that ACQUIRE acts as a minimal "acquire" operation and
     RELEASE acts as a minimal "release" operation.

A subset of the atomic operations described in atomic_t.txt have ACQUIRE and
RELEASE variants in addition to fully-ordered and relaxed (no barrier
semantics) definitions.  For compound atomics performing both a load and a
store, ACQUIRE semantics apply only to the load and RELEASE semantics apply
only to the store portion of the operation.

Memory barriers are only required where there's a possibility of interaction
between two CPUs or between a CPU and a device.  If it can be guaranteed that
there won't be any such interaction in any particular piece of code, then
memory barriers are unnecessary in that piece of code.
```

```
Note that these are the _minimum_ guarantees.  Different architectures may give
more substantial guarantees, but they may _not_ be relied upon outside of arch
specific code.


WHAT MAY NOT BE ASSUMED ABOUT MEMORY BARRIERS?
----------------------------------------------

There are certain things that the Linux kernel memory barriers do not guarantee:

 (*) There is no guarantee that any of the memory accesses specified before a
     memory barrier will be _complete_ by the completion of a memory barrier
     instruction; the barrier can be considered to draw a line in that CPU's
     access queue that accesses of the appropriate type may not cross.

 (*) There is no guarantee that issuing a memory barrier on one CPU will have
     any direct effect on another CPU or any other hardware in the system.  The
     indirect effect will be the order in which the second CPU sees the effects
     of the first CPU's accesses occur, but see the next point:

 (*) There is no guarantee that a CPU will see the correct order of effects
     from a second CPU's accesses, even _if_ the second CPU uses a memory
     barrier, unless the first CPU _also_ uses a matching memory barrier (see
     the subsection on "SMP Barrier Pairing").

 (*) There is no guarantee that some intervening piece of off-the-CPU
     hardware[*] will not reorder the memory accesses.  CPU cache coherency
     mechanisms should propagate the indirect effects of a memory barrier
     between CPUs, but might not do so in order.

        [*] For information on bus mastering DMA and coherency please read:

            Documentation/driver-api/pci/pci.rst
            Documentation/core-api/dma-api-howto.rst
            Documentation/core-api/dma-api.rst


ADDRESS-DEPENDENCY BARRIERS (HISTORICAL)
----------------------------------------

As of v4.15 of the Linux kernel, an smp_mb() was added to READ_ONCE() for
DEC Alpha, which means that about the only people who need to pay attention
to this section are those working on DEC Alpha architecture-specific code
and those working on READ_ONCE() itself.  For those who need it, and for
those who are interested in the history, here is the story of
address-dependency barriers.

[!] While address dependencies are observed in both load-to-load and
load-to-store relations, address-dependency barriers are not necessary
for load-to-store situations.

The requirement of address-dependency barriers is a little subtle, and
it's not always obvious that they're needed.  To illustrate, consider the
following sequence of events:

        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE_OLD(P);
                              D = *Q;
```

```
[!] READ_ONCE_OLD() corresponds to READ_ONCE() of pre-4.15 kernel, which
doesn't imply an address-dependency barrier.

There's a clear address dependency here, and it would seem that by the end of
the sequence, Q must be either &A or &B, and that:

        (Q == &A) implies (D == 1)
        (Q == &B) implies (D == 4)

But!  CPU 2's perception of P may be updated _before_ its perception of B, thus
leading to the following situation:

        (Q == &B) and (D == 2) ????

While this may seem like a failure of coherency or causality maintenance, it
isn't, and this behaviour can be observed on certain real CPUs (such as the DEC
Alpha).

To deal with this, READ_ONCE() provides an implicit address-dependency barrier
since kernel release v4.15:

        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE(P);
                              <implicit address-dependency barrier>
                              D = *Q;

This enforces the occurrence of one of the two implications, and prevents the
third possibility from arising.


[!] Note that this extremely counterintuitive situation arises most easily on
machines with split caches, so that, for example, one cache bank processes
even-numbered cache lines and the other bank processes odd-numbered cache
lines.  The pointer P might be stored in an odd-numbered cache line, and the
variable B might be stored in an even-numbered cache line.  Then, if the
even-numbered bank of the reading CPU's cache is extremely busy while the
odd-numbered bank is idle, one can see the new value of the pointer P (&B),
but the old value of the variable B (2).


An address-dependency barrier is not required to order dependent writes
because the CPUs that the Linux kernel supports don't do writes until they
are certain (1) that the write will actually happen, (2) of the location of
the write, and (3) of the value to be written.
But please carefully read the "CONTROL DEPENDENCIES" section and the
Documentation/RCU/rcu_dereference.rst file:  The compiler can and does break
dependencies in a great many highly creative ways.

        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C = 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE_OLD(P);
                              WRITE_ONCE(*Q, 5);
```

Therefore, no address-dependency barrier is required to order the read into Q with the store into *Q. In other words, this outcome is prohibited, even without an implicit address-dependency barrier of modern READ_ONCE():

```
(Q == &B) && (B == 4)
```

Please note that this pattern should be rare. After all, the whole point of dependency ordering is to -prevent- writes to the data structure, along with the expensive cache misses associated with those writes. This pattern can be used to record rare error conditions and the like, and the CPUs' naturally occurring ordering prevents such records from being lost.

Note well that the ordering provided by an address dependency is local to the CPU containing it. See the section on "Multicopy atomicity" for more information.

The address-dependency barrier is very important to the RCU system, for example. See rcu_assign_pointer() and rcu_dereference() in include/linux/rcupdate.h. This permits the current target of an RCU'd pointer to be replaced with a new modified target, without the replacement target appearing to be incompletely initialised.

See also the subsection on "Cache Coherency" for a more thorough example.

CONTROL DEPENDENCIES

Control dependencies can be a bit tricky because current compilers do not understand them. The purpose of this section is to help you prevent the compiler's ignorance from breaking your code.

A load-load control dependency requires a full read memory barrier, not simply an (implicit) address-dependency barrier to make it work correctly. Consider the following bit of code:

```
q = READ_ONCE(a);
<implicit address-dependency barrier>
if (q) {
        /* BUG: No address dependency!!! */
        p = READ_ONCE(b);
}
```

This will not have the desired effect because there is no actual address dependency, but rather a control dependency that the CPU may short-circuit by attempting to predict the outcome in advance, so that other CPUs see the load from b as having happened before the load from a. In such a case what's actually required is:

```
q = READ_ONCE(a);
if (q) {
        <read barrier>
        p = READ_ONCE(b);
}
```

However, stores are not speculated. This means that ordering -is- provided for load-store control dependencies, as in the following example:

```
q = READ_ONCE(a);
if (q) {
        WRITE_ONCE(b, 1);
```

```
        }

Control dependencies pair normally with other types of barriers.
That said, please note that neither READ_ONCE() nor WRITE_ONCE()
are optional! Without the READ_ONCE(), the compiler might combine the
load from 'a' with other loads from 'a'.  Without the WRITE_ONCE(),
the compiler might combine the store to 'b' with other stores to 'b'.
Either can result in highly counterintuitive effects on ordering.

Worse yet, if the compiler is able to prove (say) that the value of
variable 'a' is always non-zero, it would be well within its rights
to optimize the original example by eliminating the "if" statement
as follows:

        q = a;
        b = 1;  /* BUG: Compiler and CPU can both reorder!!! */

So don't leave out the READ_ONCE().

It is tempting to try to enforce ordering on identical stores on both
branches of the "if" statement as follows:

        q = READ_ONCE(a);
        if (q) {
                barrier();
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                barrier();
                WRITE_ONCE(b, 1);
                do_something_else();
        }

Unfortunately, current compilers will transform this as follows at high
optimization levels:

        q = READ_ONCE(a);
        barrier();
        WRITE_ONCE(b, 1);  /* BUG: No ordering vs. load from a!!! */
        if (q) {
                /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
                do_something();
        } else {
                /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
                do_something_else();
        }

Now there is no conditional between the load from 'a' and the store to
'b', which means that the CPU is within its rights to reorder them:
The conditional is absolutely required, and must be present in the
assembly code even after all compiler optimizations have been applied.
Therefore, if you need ordering in this example, you need explicit
memory barriers, for example, smp_store_release():

        q = READ_ONCE(a);
        if (q) {
                smp_store_release(&b, 1);
                do_something();
        } else {
                smp_store_release(&b, 1);
                do_something_else();
        }
```

```
In contrast, without explicit memory barriers, two-legged-if control
ordering is guaranteed only when the stores differ, for example:

        q = READ_ONCE(a);
        if (q) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }

The initial READ_ONCE() is still required to prevent the compiler from
proving the value of 'a'.

In addition, you need to be careful what you do with the local variable 'q',
otherwise the compiler might be able to guess the value and again remove
the needed conditional.  For example:

        q = READ_ONCE(a);
        if (q % MAX) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }

If MAX is defined to be 1, then the compiler knows that (q % MAX) is
equal to zero, in which case the compiler is within its rights to
transform the above code into the following:

        q = READ_ONCE(a);
        WRITE_ONCE(b, 2);
        do_something_else();

Given this transformation, the CPU is not required to respect the ordering
between the load from variable 'a' and the store to variable 'b'.  It is
tempting to add a barrier(), but this does not help.  The conditional
is gone, and the barrier won't bring it back.  Therefore, if you are
relying on this ordering, you should make sure that MAX is greater than
one, perhaps as follows:

        q = READ_ONCE(a);
        BUILD_BUG_ON(MAX <= 1); /* Order load from a with store to b. */
        if (q % MAX) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }

Please note once again that the stores to 'b' differ.  If they were
identical, as noted earlier, the compiler could pull this store outside
of the 'if' statement.

You must also be careful not to rely too much on boolean short-circuit
evaluation.  Consider this example:

        q = READ_ONCE(a);
        if (q || 1 > 0)
                WRITE_ONCE(b, 1);
```

```
Because the first condition cannot fault and the second condition is
always true, the compiler can transform this example as following,
defeating control dependency:

        q = READ_ONCE(a);
        WRITE_ONCE(b, 1);

This example underscores the need to ensure that the compiler cannot
out-guess your code.  More generally, although READ_ONCE() does force
the compiler to actually emit code for a given load, it does not force
the compiler to use the results.

In addition, control dependencies apply only to the then-clause and
else-clause of the if-statement in question.  In particular, it does
not necessarily apply to code following the if-statement:

        q = READ_ONCE(a);
        if (q) {
                WRITE_ONCE(b, 1);
        } else {
                WRITE_ONCE(b, 2);
        }
        WRITE_ONCE(c, 1);  /* BUG: No ordering against the read from 'a'. */

It is tempting to argue that there in fact is ordering because the
compiler cannot reorder volatile accesses and also cannot reorder
the writes to 'b' with the condition.  Unfortunately for this line
of reasoning, the compiler might compile the two writes to 'b' as
conditional-move instructions, as in this fanciful pseudo-assembly
language:

        ld r1,a
        cmp r1,$0
        cmov,ne r4,$1
        cmov,eq r4,$2
        st r4,b
        st $1,c

A weakly ordered CPU would have no dependency of any sort between the load
from 'a' and the store to 'c'.  The control dependencies would extend
only to the pair of cmov instructions and the store depending on them.
In short, control dependencies apply only to the stores in the then-clause
and else-clause of the if-statement in question (including functions
invoked by those two clauses), not to code following that if-statement.


Note well that the ordering provided by a control dependency is local
to the CPU containing it.  See the section on "Multicopy atomicity"
for more information.


In summary:

  (*) Control dependencies can order prior loads against later stores.
      However, they do -not- guarantee any other sort of ordering:
      Not prior loads against later loads, nor prior stores against
      later anything.  If you need these other forms of ordering,
      use smp_rmb(), smp_wmb(), or, in the case of prior stores and
      later loads, smp_mb().

  (*) If both legs of the "if" statement begin with identical stores to
      the same variable, then those stores must be ordered, either by
```

preceding both of them with smp_mb() or by using smp_store_release() to carry out the stores. Please note that it is -not- sufficient to use barrier() at beginning of each leg of the "if" statement because, as shown by the example above, optimizing compilers can destroy the control dependency while respecting the letter of the barrier() law.

(*) Control dependencies require at least one run-time conditional between the prior load and the subsequent store, and this conditional must involve the prior load. If the compiler is able to optimize the conditional away, it will have also optimized away the ordering. Careful use of READ_ONCE() and WRITE_ONCE() can help to preserve the needed conditional.

(*) Control dependencies require that the compiler avoid reordering the dependency into nonexistence. Careful use of READ_ONCE() or atomic{,64}_read() can help to preserve your control dependency. Please see the COMPILER BARRIER section for more information.

(*) Control dependencies apply only to the then-clause and else-clause of the if-statement containing the control dependency, including any functions that these two clauses call. Control dependencies do -not- apply to code following the if-statement containing the control dependency.

(*) Control dependencies pair normally with other types of barriers.

(*) Control dependencies do -not- provide multicopy atomicity. If you need all the CPUs to see a given store at the same time, use smp_mb().

(*) Compilers do not understand control dependencies. It is therefore your job to ensure that they do not break your code.

SMP BARRIER PAIRING

When dealing with CPU-CPU interactions, certain types of memory barrier should always be paired. A lack of appropriate pairing is almost certainly an error.

General barriers pair with each other, though they also pair with most other types of barriers, albeit without multicopy atomicity. An acquire barrier pairs with a release barrier, but both may also pair with other barriers, including of course general barriers. A write barrier pairs with an address-dependency barrier, a control dependency, an acquire barrier, a release barrier, a read barrier, or a general barrier. Similarly a read barrier, control dependency, or an address-dependency barrier pairs with a write barrier, an acquire barrier, a release barrier, or a general barrier:

```
CPU 1                 CPU 2
===============       ===============
WRITE_ONCE(a, 1);
<write barrier>
WRITE_ONCE(b, 2);     x = READ_ONCE(b);
                      <read barrier>
                      y = READ_ONCE(a);
```

Or:

```
CPU 1                 CPU 2
===============       ===============================
a = 1;
```

```
        <write barrier>
        WRITE_ONCE(b, &a);    x = READ_ONCE(b);
                              <implicit address-dependency barrier>
                              y = *x;
```

Or even:

```
        CPU 1                 CPU 2
        ===============       ===============================
        r1 = READ_ONCE(y);
        <general barrier>
        WRITE_ONCE(x, 1);     if (r2 = READ_ONCE(x)) {
                                 <implicit control dependency>
                                 WRITE_ONCE(y, 1);
                              }

        assert(r1 == 0 || r2 == 0);
```

Basically, the read barrier always has to be there, even though it can be of the "weaker" type.

[!] Note that the stores before the write barrier would normally be expected to match the loads after the read barrier or the address-dependency barrier, and vice versa:

```
        CPU 1                               CPU 2
        ===================                 ===================
        WRITE_ONCE(a, 1);    }----   --->{  v = READ_ONCE(c);
        WRITE_ONCE(b, 2);    }    \ /    {  w = READ_ONCE(d);
        <write barrier>            \        <read barrier>
        WRITE_ONCE(c, 3);    }    / \    {  x = READ_ONCE(a);
        WRITE_ONCE(d, 4);    }----   --->{  y = READ_ONCE(b);
```

EXAMPLES OF MEMORY BARRIER SEQUENCES

Firstly, write barriers act as partial orderings on store operations. Consider the following sequence of events:

```
        CPU 1
        =======================
        STORE A = 1
        STORE B = 2
        STORE C = 3
        <write barrier>
        STORE D = 4
        STORE E = 5
```

This sequence of events is committed to the memory coherence system in an order that the rest of the system might perceive as the unordered set of { STORE A, STORE B, STORE C } all occurring before the unordered set of { STORE D, STORE E }:

```
        +-------+       :      :
        |       |       +------+
        |       |------>| C=3  |     }     /\
        |       |  :    +------+     }-----  \  -----> Events perceptible to
        |       |  :    | A=1  |     }        \/       the rest of the system
        |       |  :    +------+     }
        | CPU 1 |  :    | B=2  |     }
        |       |       +------+     }
        |       |   wwwwwwwwwwwwwwww }   <--- At this point the write barrier
```

```
        |       |       +------+     }        requires all stores prior to the
        |       |  :    | E=5  |     }        barrier to be committed before
        |       |  :    +------+     }        further stores may take place
        |       |------>| D=4  |     }
        |       |       +------+
        +-------+       :      :
                           |
                           | Sequence in which stores are committed to the
                           | memory system by CPU 1
                           V
```

Secondly, address-dependency barriers act as partial orderings on address-dependent loads. Consider the following sequence of events:

```
        CPU 1                   CPU 2
        ======================= =======================
                { B = 7; X = 9; Y = 8; C = &Y }
        STORE A = 1
        STORE B = 2
        <write barrier>
        STORE C = &B            LOAD X
        STORE D = 4             LOAD C (gets &B)
                                LOAD *C (reads B)
```

Without intervention, CPU 2 may perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+  | Sequence of update
        |       |------>| B=2  |-----       --->| Y->8  |  | of perception on
        |       |  :    +------+     \          +-------+  | CPU 2
        | CPU 1 |  :    | A=1  |      \     --->| C->&Y |  V
        |       |       +------+       |        +-------+
        |       |   wwwwwwwwwwwwwwww   |        :       :
        |       |       +------+       |        :       :
        |       |  :    | C=&B |---    |        :       :       +-------+
        |       |  :    +------+   \   |        +-------+       |       |
        |       |------>| D=4  |    ----------->| C->&B |------>|       |
        |       |       +------+       |        +-------+       |       |
        +-------+       :      :       |        :       :       |       |
                                       |        :       :       |       |
                                       |        :       :       | CPU 2 |
                                       |        +-------+       |       |
            Apparently incorrect --->  |        | B->7  |------>|       |
            perception of B (!)        |        +-------+       |       |
                                       |        :       :       |       |
                                       |        +-------+       |       |
            The load of X holds --->    \       | X->9  |------>|       |
            up the maintenance           \      +-------+       |       |
            of coherence of B             ----->| B->2  |       +-------+
                                                +-------+
                                                :       :
```

In the above example, CPU 2 perceives that B is 7, despite the load of *C (which would be B) coming after the LOAD of C.

If, however, an address-dependency barrier were to be placed between the load of C and the load of *C (ie: B) on CPU 2:

```
        CPU 1                   CPU 2
        ======================= =======================
```

```
                { B = 7; X = 9; Y = 8; C = &Y }
        STORE A = 1
        STORE B = 2
        <write barrier>
        STORE C = &B            LOAD X
        STORE D = 4             LOAD C (gets &B)
                                <address-dependency barrier>
                                LOAD *C (reads B)
```

then the following will occur:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| B=2  |-----       --->| Y->8  |
        |       |  :    +------+     \          +-------+
        | CPU 1 |  :    | A=1  |      \     --->| C->&Y |
        |       |       +------+       |        +-------+
        |       |   wwwwwwwwwwwwwwww   |        :       :
        |       |       +------+       |        :       :
        |       |  :    | C=&B |---    |        :       :       +-------+
        |       |  :    +------+   \   |        +-------+       |       |
        |       |------>| D=4  |    ----------->| C->&B |------>|       |
        |       |       +------+       |        +-------+       |       |
        +-------+       :      :       |        :       :       |       |
                                       |        :       :       |       |
                                       |        :       :       | CPU 2 |
                                       |        +-------+       |       |
                                       |        | X->9  |------>|       |
                                       |        +-------+       |       |
          Makes sure all effects --->   \   aaaaaaaaaaaaaaaaa   |       |
          prior to the store of C        \      +-------+       |       |
          are perceptible to              ----->| B->2  |------>|       |
          subsequent loads                      +-------+       |       |
                                                :       :       +-------+
```

And thirdly, a read barrier acts as a partial order on loads. Consider the following sequence of events:

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                LOAD A
```

Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       | A->0  |------>|       |
```

```
                                         |       +-------+       |       |
                                         |       :       :       +-------+
                                          \      :       :
                                           \     +-------+
                                            ---->| A->1  |
                                                 +-------+
                                                 :       :
```

If, however, a read barrier were to be placed between the load of B and the load of A on CPU 2:

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                <read barrier>
                                LOAD A
```

then the partial ordering imposed by CPU 1 will be perceived correctly by CPU 2:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                        |       :       :       |       |
          At this point the read ---->   \  rrrrrrrrrrrrrrrrr   |       |
          barrier causes all effects      \     +-------+       |       |
          prior to the storage of B        ---->| A->1  |------>|       |
          to be perceptible to CPU 2            +-------+       |       |
                                                :       :       +-------+
```

To illustrate this more completely, consider what could happen if the code contained a load of A either side of the read barrier:

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                LOAD A [first load of A]
                                <read barrier>
                                LOAD A [second load of A]
```

Even though the two loads of A both occur after the load of B, they may both come up with different values:

```
        +-------+       :      :                :       :
```

```
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                        |       :       :       |       |
                                        |       +-------+       |       |
                                        |       | A->0  |------>| 1st   |
                                        |       +-------+       |       |
          At this point the read ---->   \  rrrrrrrrrrrrrrrrr   |       |
          barrier causes all effects      \     +-------+       |       |
          prior to the storage of B        ---->| A->1  |------>| 2nd   |
          to be perceptible to CPU 2            +-------+       |       |
                                                :       :       +-------+
```

But it may be that the update to A from CPU 1 becomes perceptible to CPU 2
before the read barrier completes anyway:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                         \      :       :       |       |
                                          \     +-------+       |       |
                                           ---->| A->1  |------>| 1st   |
                                                +-------+       |       |
                                            rrrrrrrrrrrrrrrrr   |       |
                                                +-------+       |       |
                                                | A->1  |------>| 2nd   |
                                                +-------+       |       |
                                                :       :       +-------+
```

The guarantee is that the second load will always come up with A == 1 if the load of B came up with B == 2. No such guarantee exists for the first load of A; that may come up with either A == 0 or A == 1.

READ MEMORY BARRIERS VS LOAD SPECULATION

Many CPUs speculate with loads: that is they see that they will need to load an item from memory, and they find a time where they're not using the bus for any other loads, and so do the load in advance - even though they haven't actually got to that point in the instruction execution flow yet. This permits the actual load instruction to potentially complete immediately because the CPU already has the value to hand.

```
It may turn out that the CPU didn't actually need the value - perhaps because a
branch circumvented the load - in which case it can discard the value or just
cache it for later use.

Consider:

        CPU 1                   CPU 2
        ======================= =======================
                                LOAD B
                                DIVIDE          } Divide instructions generally
                                DIVIDE          } take a long time to perform
                                LOAD A

Which might appear as this:

                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
        Once the divisions are complete -->     :       :   ~-->|       |
        the CPU can then perform the            :       :       |       |
        LOAD with immediate effect              :       :       +-------+


Placing a read barrier or an address-dependency barrier just before the second
load:

        CPU 1                   CPU 2
        ======================= =======================
                                LOAD B
                                DIVIDE
                                DIVIDE
                                <read barrier>
                                LOAD A

will force any value speculatively obtained to be reconsidered to an extent
dependent on the type of barrier used.  If there was no change made to the
speculated memory location, then the speculated value will just be used:

                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
                                                :       :   ~   |       |
                                            rrrrrrrrrrrrrrrr~   |       |
                                                :       :   ~   |       |
                                                :       :   ~-->|       |
                                                :       :       |       |
                                                :       :       +-------+
```

but if there was an update or an invalidation from another CPU pending, then
the speculation will be cancelled and the value reloaded:

```
                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
                                                :       :   ~   |       |
                                            rrrrrrrrrrrrrrrr    |       |
                                                +-------+       |       |
        The speculation is discarded --->   --->| A->1  |------>|       |
        and an updated value is                 +-------+       |       |
        retrieved                               :       :       +-------+
```

MULTICOPY ATOMICITY

Multicopy atomicity is a deeply intuitive notion about ordering that is
not always provided by real computer systems, namely that a given store
becomes visible at the same time to all CPUs, or, alternatively, that all
CPUs agree on the order in which all stores become visible. However,
support of full multicopy atomicity would rule out valuable hardware
optimizations, so a weaker form called ``other multicopy atomicity''
instead guarantees only that a given store becomes visible at the same
time to all -other- CPUs. The remainder of this document discusses this
weaker form, but for brevity will call it simply ``multicopy atomicity''.

The following example demonstrates multicopy atomicity:

```
        CPU 1                   CPU 2                   CPU 3
        ======================= ======================= =======================
                { X = 0, Y = 0 }
        STORE X=1               r1=LOAD X (reads 1)     LOAD Y (reads 1)
                                <general barrier>       <read barrier>
                                STORE Y=r1              LOAD X
```

Suppose that CPU 2's load from X returns 1, which it then stores to Y,
and CPU 3's load from Y returns 1. This indicates that CPU 1's store
to X precedes CPU 2's load from X and that CPU 2's store to Y precedes
CPU 3's load from Y. In addition, the memory barriers guarantee that
CPU 2 executes its load before its store, and CPU 3 loads from Y before
it loads from X. The question is then "Can CPU 3's load from X return 0?"

Because CPU 3's load from X in some sense comes after CPU 2's load, it
is natural to expect that CPU 3's load from X must therefore return 1.
This expectation follows from multicopy atomicity: if a load executing
on CPU B follows a load from the same variable executing on CPU A (and
CPU A did not originally store the value which it read), then on
multicopy-atomic systems, CPU B's load must return either the same value
that CPU A's load did or some later value. However, the Linux kernel
does not require systems to be multicopy atomic.

The use of a general memory barrier in the example above compensates

```
for any lack of multicopy atomicity.  In the example, if CPU 2's load
from X returns 1 and CPU 3's load from Y returns 1, then CPU 3's load
from X must indeed also return 1.

However, dependencies, read barriers, and write barriers are not always
able to compensate for non-multicopy atomicity.  For example, suppose
that CPU 2's general barrier is removed from the above example, leaving
only the data dependency shown below:

        CPU 1                   CPU 2                   CPU 3
        ======================= ======================= =======================
                { X = 0, Y = 0 }
        STORE X=1               r1=LOAD X (reads 1)     LOAD Y (reads 1)
                                <data dependency>       <read barrier>
                                STORE Y=r1              LOAD X (reads 0)

This substitution allows non-multicopy atomicity to run rampant: in
this example, it is perfectly legal for CPU 2's load from X to return 1,
CPU 3's load from Y to return 1, and its load from X to return 0.

The key point is that although CPU 2's data dependency orders its load
and store, it does not guarantee to order CPU 1's store.  Thus, if this
example runs on a non-multicopy-atomic system where CPUs 1 and 2 share a
store buffer or a level of cache, CPU 2 might have early access to CPU 1's
writes.  General barriers are therefore required to ensure that all CPUs
agree on the combined order of multiple accesses.

General barriers can compensate not only for non-multicopy atomicity,
but can also generate additional ordering that can ensure that -all-
CPUs will perceive the same order of -all- operations.  In contrast, a
chain of release-acquire pairs do not provide this additional ordering,
which means that only those CPUs on the chain are guaranteed to agree
on the combined order of the accesses.  For example, switching to C code
in deference to the ghost of Herman Hollerith:

        int u, v, x, y, z;

        void cpu0(void)
        {
                r0 = smp_load_acquire(&x);
                WRITE_ONCE(u, 1);
                smp_store_release(&y, 1);
        }

        void cpu1(void)
        {
                r1 = smp_load_acquire(&y);
                r4 = READ_ONCE(v);
                r5 = READ_ONCE(u);
                smp_store_release(&z, 1);
        }

        void cpu2(void)
        {
                r2 = smp_load_acquire(&z);
                smp_store_release(&x, 1);
        }

        void cpu3(void)
        {
                WRITE_ONCE(v, 1);
                smp_mb();
                r3 = READ_ONCE(u);
```

```
        }

Because cpu0(), cpu1(), and cpu2() participate in a chain of
smp_store_release()/smp_load_acquire() pairs, the following outcome
is prohibited:

        r0 == 1 && r1 == 1 && r2 == 1

Furthermore, because of the release-acquire relationship between cpu0()
and cpu1(), cpu1() must see cpu0()'s writes, so that the following
outcome is prohibited:

        r1 == 1 && r5 == 0

However, the ordering provided by a release-acquire chain is local
to the CPUs participating in that chain and does not apply to cpu3(),
at least aside from stores.  Therefore, the following outcome is possible:

        r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0

As an aside, the following outcome is also possible:

        r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0 && r5 == 1

Although cpu0(), cpu1(), and cpu2() will see their respective reads and
writes in order, CPUs not involved in the release-acquire chain might
well disagree on the order.  This disagreement stems from the fact that
the weak memory-barrier instructions used to implement smp_load_acquire()
and smp_store_release() are not required to order prior stores against
subsequent loads in all cases.  This means that cpu3() can see cpu0()'s
store to u as happening -after- cpu1()'s load from v, even though
both cpu0() and cpu1() agree that these two operations occurred in the
intended order.

However, please keep in mind that smp_load_acquire() is not magic.
In particular, it simply reads from its argument with ordering.  It does
-not- ensure that any particular value will be read.  Therefore, the
following outcome is possible:

        r0 == 0 && r1 == 0 && r2 == 0 && r5 == 0

Note that this outcome can happen even on a mythical sequentially
consistent system where nothing is ever reordered.

To reiterate, if your code requires full ordering of all operations,
use general barriers throughout.


========================
EXPLICIT KERNEL BARRIERS
========================

The Linux kernel has a variety of different barriers that act at different
levels:

  (*) Compiler barrier.

  (*) CPU memory barriers.


COMPILER BARRIER
----------------
```

```
The Linux kernel has an explicit compiler barrier function that prevents the
compiler from moving the memory accesses either side of it to the other side:

        barrier();

This is a general barrier -- there are no read-read or write-write
variants of barrier().  However, READ_ONCE() and WRITE_ONCE() can be
thought of as weak forms of barrier() that affect only the specific
accesses flagged by the READ_ONCE() or WRITE_ONCE().

The barrier() function has the following effects:

 (*) Prevents the compiler from reordering accesses following the
     barrier() to precede any accesses preceding the barrier().
     One example use for this property is to ease communication between
     interrupt-handler code and the code that was interrupted.

 (*) Within a loop, forces the compiler to load the variables used
     in that loop's conditional on each pass through that loop.

The READ_ONCE() and WRITE_ONCE() functions can prevent any number of
optimizations that, while perfectly safe in single-threaded code, can
be fatal in concurrent code.  Here are some examples of these sorts
of optimizations:

 (*) The compiler is within its rights to reorder loads and stores
     to the same variable, and in some cases, the CPU is within its
     rights to reorder loads to the same variable.  This means that
     the following code:

        a[0] = x;
        a[1] = x;

     Might result in an older value of x stored in a[1] than in a[0].
     Prevent both the compiler and the CPU from doing this as follows:

        a[0] = READ_ONCE(x);
        a[1] = READ_ONCE(x);

     In short, READ_ONCE() and WRITE_ONCE() provide cache coherence for
     accesses from multiple CPUs to a single variable.

 (*) The compiler is within its rights to merge successive loads from
     the same variable.  Such merging can cause the compiler to "optimize"
     the following code:

        while (tmp = a)
                do_something_with(tmp);

     into the following code, which, although in some sense legitimate
     for single-threaded code, is almost certainly not what the developer
     intended:

        if (tmp = a)
                for (;;)
                        do_something_with(tmp);

     Use READ_ONCE() to prevent the compiler from doing this to you:

        while (tmp = READ_ONCE(a))
                do_something_with(tmp);

 (*) The compiler is within its rights to reload a variable, for example,
```

```
     in cases where high register pressure prevents the compiler from
     keeping all data of interest in registers.  The compiler might
     therefore optimize the variable 'tmp' out of our previous example:

        while (tmp = a)
                do_something_with(tmp);

     This could result in the following code, which is perfectly safe in
     single-threaded code, but can be fatal in concurrent code:

        while (a)
                do_something_with(a);

     For example, the optimized version of this code could result in
     passing a zero to do_something_with() in the case where the variable
     a was modified by some other CPU between the "while" statement and
     the call to do_something_with().

     Again, use READ_ONCE() to prevent the compiler from doing this:

        while (tmp = READ_ONCE(a))
                do_something_with(tmp);

     Note that if the compiler runs short of registers, it might save
     tmp onto the stack.  The overhead of this saving and later restoring
     is why compilers reload variables.  Doing so is perfectly safe for
     single-threaded code, so you need to tell the compiler about cases
     where it is not safe.

 (*) The compiler is within its rights to omit a load entirely if it knows
     what the value will be.  For example, if the compiler can prove that
     the value of variable 'a' is always zero, it can optimize this code:

        while (tmp = a)
                do_something_with(tmp);

     Into this:

        do { } while (0);

     This transformation is a win for single-threaded code because it
     gets rid of a load and a branch.  The problem is that the compiler
     will carry out its proof assuming that the current CPU is the only
     one updating variable 'a'.  If variable 'a' is shared, then the
     compiler's proof will be erroneous.  Use READ_ONCE() to tell the
     compiler that it doesn't know as much as it thinks it does:

        while (tmp = READ_ONCE(a))
                do_something_with(tmp);

     But please note that the compiler is also closely watching what you
     do with the value after the READ_ONCE().  For example, suppose you
     do the following and MAX is a preprocessor macro with the value 1:

        while ((tmp = READ_ONCE(a)) % MAX)
                do_something_with(tmp);

     Then the compiler knows that the result of the "%" operator applied
     to MAX will always be zero, again allowing the compiler to optimize
     the code into near-nonexistence.  (It will still load from the
     variable 'a'.)

 (*) Similarly, the compiler is within its rights to omit a store entirely
```

```
    if it knows that the variable already has the value being stored.
    Again, the compiler assumes that the current CPU is the only one
    storing into the variable, which can cause the compiler to do the
    wrong thing for shared variables.  For example, suppose you have
    the following:

       a = 0;
       ... Code that does not store to variable a ...
       a = 0;

    The compiler sees that the value of variable 'a' is already zero, so
    it might well omit the second store.  This would come as a fatal
    surprise if some other CPU might have stored to variable 'a' in the
    meantime.

    Use WRITE_ONCE() to prevent the compiler from making this sort of
    wrong guess:

       WRITE_ONCE(a, 0);
       ... Code that does not store to variable a ...
       WRITE_ONCE(a, 0);

(*) The compiler is within its rights to reorder memory accesses unless
    you tell it not to.  For example, consider the following interaction
    between process-level code and an interrupt handler:

       void process_level(void)
       {
               msg = get_message();
               flag = true;
       }

       void interrupt_handler(void)
       {
               if (flag)
                       process_message(msg);
       }

    There is nothing to prevent the compiler from transforming
    process_level() to the following, in fact, this might well be a
    win for single-threaded code:

       void process_level(void)
       {
               flag = true;
               msg = get_message();
       }

    If the interrupt occurs between these two statement, then
    interrupt_handler() might be passed a garbled msg.  Use WRITE_ONCE()
    to prevent this as follows:

       void process_level(void)
       {
               WRITE_ONCE(msg, get_message());
               WRITE_ONCE(flag, true);
       }

       void interrupt_handler(void)
       {
               if (READ_ONCE(flag))
                       process_message(READ_ONCE(msg));
       }
```

```
     Note that the READ_ONCE() and WRITE_ONCE() wrappers in
     interrupt_handler() are needed if this interrupt handler can itself
     be interrupted by something that also accesses 'flag' and 'msg',
     for example, a nested interrupt or an NMI.  Otherwise, READ_ONCE()
     and WRITE_ONCE() are not needed in interrupt_handler() other than
     for documentation purposes.  (Note also that nested interrupts
     do not typically occur in modern Linux kernels, in fact, if an
     interrupt handler returns with interrupts enabled, you will get a
     WARN_ONCE() splat.)

     You should assume that the compiler can move READ_ONCE() and
     WRITE_ONCE() past code not containing READ_ONCE(), WRITE_ONCE(),
     barrier(), or similar primitives.

     This effect could also be achieved using barrier(), but READ_ONCE()
     and WRITE_ONCE() are more selective:  With READ_ONCE() and
     WRITE_ONCE(), the compiler need only forget the contents of the
     indicated memory locations, while with barrier() the compiler must
     discard the value of all memory locations that it has currently
     cached in any machine registers.  Of course, the compiler must also
     respect the order in which the READ_ONCE()s and WRITE_ONCE()s occur,
     though the CPU of course need not do so.

 (*) The compiler is within its rights to invent stores to a variable,
     as in the following example:

        if (a)
                b = a;
        else
                b = 42;

     The compiler might save a branch by optimizing this as follows:

        b = 42;
        if (a)
                b = a;

     In single-threaded code, this is not only safe, but also saves
     a branch.  Unfortunately, in concurrent code, this optimization
     could cause some other CPU to see a spurious value of 42 -- even
     if variable 'a' was never zero -- when loading variable 'b'.
     Use WRITE_ONCE() to prevent this as follows:

        if (a)
                WRITE_ONCE(b, a);
        else
                WRITE_ONCE(b, 42);

     The compiler can also invent loads.  These are usually less
     damaging, but they can result in cache-line bouncing and thus in
     poor performance and scalability.  Use READ_ONCE() to prevent
     invented loads.

 (*) For aligned memory locations whose size allows them to be accessed
     with a single memory-reference instruction, prevents "load tearing"
     and "store tearing," in which a single large access is replaced by
     multiple smaller accesses.  For example, given an architecture having
     16-bit store instructions with 7-bit immediate fields, the compiler
     might be tempted to use two 16-bit store-immediate instructions to
     implement the following 32-bit store:

        p = 0x00010002;
```

```
     Please note that GCC really does use this sort of optimization,
     which is not surprising given that it would likely take more
     than two instructions to build the constant and then store it.
     This optimization can therefore be a win in single-threaded code.
     In fact, a recent bug (since fixed) caused GCC to incorrectly use
     this optimization in a volatile store.  In the absence of such bugs,
     use of WRITE_ONCE() prevents store tearing in the following example:

        WRITE_ONCE(p, 0x00010002);

     Use of packed structures can also result in load and store tearing,
     as in this example:

        struct __attribute__((__packed__)) foo {
                short a;
                int b;
                short c;
        };
        struct foo foo1, foo2;
        ...

        foo2.a = foo1.a;
        foo2.b = foo1.b;
        foo2.c = foo1.c;

     Because there are no READ_ONCE() or WRITE_ONCE() wrappers and no
     volatile markings, the compiler would be well within its rights to
     implement these three assignment statements as a pair of 32-bit
     loads followed by a pair of 32-bit stores.  This would result in
     load tearing on 'foo1.b' and store tearing on 'foo2.b'.  READ_ONCE()
     and WRITE_ONCE() again prevent tearing in this example:

        foo2.a = foo1.a;
        WRITE_ONCE(foo2.b, READ_ONCE(foo1.b));
        foo2.c = foo1.c;

All that aside, it is never necessary to use READ_ONCE() and
WRITE_ONCE() on a variable that has been marked volatile.  For example,
because 'jiffies' is marked volatile, it is never necessary to
say READ_ONCE(jiffies).  The reason for this is that READ_ONCE() and
WRITE_ONCE() are implemented as volatile casts, which has no effect when
its argument is already marked volatile.

Please note that these compiler barriers have no direct effect on the CPU,
which may then reorder things however it wishes.


CPU MEMORY BARRIERS
-------------------

The Linux kernel has seven basic CPU memory barriers:

        TYPE                    MANDATORY       SMP CONDITIONAL
        ======================= =============== ===============
        GENERAL                 mb()            smp_mb()
        WRITE                   wmb()           smp_wmb()
        READ                    rmb()           smp_rmb()
        ADDRESS DEPENDENCY                      READ_ONCE()


All memory barriers except the address-dependency barriers imply a compiler
barrier.  Address dependencies do not impose any additional compiler ordering.
```

```
Aside: In the case of address dependencies, the compiler would be expected
to issue the loads in the correct order (eg. `a[b]` would have to load
the value of b before loading a[b]), however there is no guarantee in
the C specification that the compiler may not speculate the value of b
(eg. is equal to 1) and load a[b] before b (eg. tmp = a[1]; if (b != 1)
tmp = a[b]; ).  There is also the problem of a compiler reloading b after
having loaded a[b], thus having a newer copy of b than a[b].  A consensus
has not yet been reached about these problems, however the READ_ONCE()
macro is a good place to start looking.

SMP memory barriers are reduced to compiler barriers on uniprocessor compiled
systems because it is assumed that a CPU will appear to be self-consistent,
and will order overlapping accesses correctly with respect to itself.
However, see the subsection on "Virtual Machine Guests" below.

[!] Note that SMP memory barriers _must_ be used to control the ordering of
references to shared memory on SMP systems, though the use of locking instead
is sufficient.

Mandatory barriers should not be used to control SMP effects, since mandatory
barriers impose unnecessary overhead on both SMP and UP systems. They may,
however, be used to control MMIO effects on accesses through relaxed memory I/O
windows.  These barriers are required even on non-SMP systems as they affect
the order in which memory operations appear to a device by prohibiting both the
compiler and the CPU from reordering them.


There are some more advanced barrier functions:

 (*) smp_store_mb(var, value)

     This assigns the value to the variable and then inserts a full memory
     barrier after it.  It isn't guaranteed to insert anything more than a
     compiler barrier in a UP compilation.


 (*) smp_mb__before_atomic();
 (*) smp_mb__after_atomic();

     These are for use with atomic RMW functions that do not imply memory
     barriers, but where the code needs a memory barrier. Examples for atomic
     RMW functions that do not imply a memory barrier are e.g. add,
     subtract, (failed) conditional operations, _relaxed functions,
     but not atomic_read or atomic_set. A common example where a memory
     barrier may be required is when atomic ops are used for reference
     counting.

     These are also used for atomic RMW bitop functions that do not imply a
     memory barrier (such as set_bit and clear_bit).

     As an example, consider a piece of code that marks an object as being dead
     and then decrements the object's reference count:

        obj->dead = 1;
        smp_mb__before_atomic();
        atomic_dec(&obj->ref_count);

     This makes sure that the death mark on the object is perceived to be set
     *before* the reference counter is decremented.

     See Documentation/atomic_{t,bitops}.txt for more information.
```

```
(*) dma_wmb();
(*) dma_rmb();
(*) dma_mb();

    These are for use with consistent memory to guarantee the ordering
    of writes or reads of shared memory accessible to both the CPU and a
    DMA capable device. See Documentation/core-api/dma-api.rst file for more
    information about consistent memory.

    For example, consider a device driver that shares memory with a device
    and uses a descriptor status value to indicate if the descriptor belongs
    to the device or the CPU, and a doorbell to notify it when new
    descriptors are available:

       if (desc->status != DEVICE_OWN) {
               /* do not read data until we own descriptor */
               dma_rmb();

               /* read/modify data */
               read_data = desc->data;
               desc->data = write_data;

               /* flush modifications before status update */
               dma_wmb();

               /* assign ownership */
               desc->status = DEVICE_OWN;

               /* Make descriptor status visible to the device followed by
                * notify device of new descriptor
                */
               writel(DESC_NOTIFY, doorbell);
       }

    The dma_rmb() allows us to guarantee that the device has released ownership
    before we read the data from the descriptor, and the dma_wmb() allows
    us to guarantee the data is written to the descriptor before the device
    can see it now has ownership.  The dma_mb() implies both a dma_rmb() and
    a dma_wmb().

    Note that the dma_*() barriers do not provide any ordering guarantees for
    accesses to MMIO regions.  See the later "KERNEL I/O BARRIER EFFECTS"
    subsection for more information about I/O accessors and MMIO ordering.

(*) pmem_wmb();

    This is for use with persistent memory to ensure that stores for which
    modifications are written to persistent storage reached a platform
    durability domain.

    For example, after a non-temporal write to pmem region, we use pmem_wmb()
    to ensure that stores have reached a platform durability domain. This ensures
    that stores have updated persistent storage before any data access or
    data transfer caused by subsequent instructions is initiated. This is
    in addition to the ordering done by wmb().

    For load from persistent memory, existing read memory barriers are sufficient
    to ensure read ordering.

(*) io_stop_wc();

    For memory accesses with write-combining attributes (e.g. those returned
```

```
     by ioremap_wc()), the CPU may wait for prior accesses to be merged with
     subsequent ones. io_stop_wc() can be used to prevent the merging of
     write-combining memory accesses before this macro with those after it when
     such wait has performance implications.

===============================
IMPLICIT KERNEL MEMORY BARRIERS
===============================

Some of the other functions in the linux kernel imply memory barriers, amongst
which are locking and scheduling functions.

This specification is a _minimum_ guarantee; any particular architecture may
provide more substantial guarantees, but these may not be relied upon outside
of arch specific code.


LOCK ACQUISITION FUNCTIONS
--------------------------

The Linux kernel has a number of locking constructs:

 (*) spin locks
 (*) R/W spin locks
 (*) mutexes
 (*) semaphores
 (*) R/W semaphores

In all cases there are variants on "ACQUIRE" operations and "RELEASE" operations
for each construct.  These operations all imply certain barriers:

 (1) ACQUIRE operation implication:

     Memory operations issued after the ACQUIRE will be completed after the
     ACQUIRE operation has completed.

     Memory operations issued before the ACQUIRE may be completed after
     the ACQUIRE operation has completed.

 (2) RELEASE operation implication:

     Memory operations issued before the RELEASE will be completed before the
     RELEASE operation has completed.

     Memory operations issued after the RELEASE may be completed before the
     RELEASE operation has completed.

 (3) ACQUIRE vs ACQUIRE implication:

     All ACQUIRE operations issued before another ACQUIRE operation will be
     completed before that ACQUIRE operation.

 (4) ACQUIRE vs RELEASE implication:

     All ACQUIRE operations issued before a RELEASE operation will be
     completed before the RELEASE operation.

 (5) Failed conditional ACQUIRE implication:

     Certain locking variants of the ACQUIRE operation may fail, either due to
     being unable to get the lock immediately, or due to receiving an unblocked
     signal while asleep waiting for the lock to become available.  Failed
     locks do not imply any sort of barrier.
```

```
[!] Note: one of the consequences of lock ACQUIREs and RELEASEs being only
one-way barriers is that the effects of instructions outside of a critical
section may seep into the inside of the critical section.

An ACQUIRE followed by a RELEASE may not be assumed to be full memory barrier
because it is possible for an access preceding the ACQUIRE to happen after the
ACQUIRE, and an access following the RELEASE to happen before the RELEASE, and
the two accesses can themselves then cross:

        *A = a;
        ACQUIRE M
        RELEASE M
        *B = b;

may occur as:

        ACQUIRE M, STORE *B, STORE *A, RELEASE M

When the ACQUIRE and RELEASE are a lock acquisition and release,
respectively, this same reordering can occur if the lock's ACQUIRE and
RELEASE are to the same lock variable, but only from the perspective of
another CPU not holding that lock.  In short, a ACQUIRE followed by an
RELEASE may -not- be assumed to be a full memory barrier.

Similarly, the reverse case of a RELEASE followed by an ACQUIRE does
not imply a full memory barrier.  Therefore, the CPU's execution of the
critical sections corresponding to the RELEASE and the ACQUIRE can cross,
so that:

        *A = a;
        RELEASE M
        ACQUIRE N
        *B = b;

could occur as:

        ACQUIRE N, STORE *B, STORE *A, RELEASE M

It might appear that this reordering could introduce a deadlock.
However, this cannot happen because if such a deadlock threatened,
the RELEASE would simply complete, thereby avoiding the deadlock.

        Why does this work?

        One key point is that we are only talking about the CPU doing
        the reordering, not the compiler.  If the compiler (or, for
        that matter, the developer) switched the operations, deadlock
        -could- occur.

        But suppose the CPU reordered the operations.  In this case,
        the unlock precedes the lock in the assembly code.  The CPU
        simply elected to try executing the later lock operation first.
        If there is a deadlock, this lock operation will simply spin (or
        try to sleep, but more on that later).  The CPU will eventually
        execute the unlock operation (which preceded the lock operation
        in the assembly code), which will unravel the potential deadlock,
        allowing the lock operation to succeed.

        But what if the lock is a sleeplock?  In that case, the code will
        try to enter the scheduler, where it will eventually encounter
        a memory barrier, which will force the earlier unlock operation
        to complete, again unraveling the deadlock.  There might be
```

```
        a sleep-unlock race, but the locking primitive needs to resolve
        such races properly in any case.

Locks and semaphores may not provide any guarantee of ordering on UP compiled
systems, and so cannot be counted on in such a situation to actually achieve
anything at all - especially with respect to I/O accesses - unless combined
with interrupt disabling operations.

See also the section on "Inter-CPU acquiring barrier effects".


As an example, consider the following:

        *A = a;
        *B = b;
        ACQUIRE
        *C = c;
        *D = d;
        RELEASE
        *E = e;
        *F = f;

The following sequence of events is acceptable:

        ACQUIRE, {*F,*A}, *E, {*C,*D}, *B, RELEASE

        [+] Note that {*F,*A} indicates a combined access.

But none of the following are:

        {*F,*A}, *B,    ACQUIRE, *C, *D,        RELEASE, *E
        *A, *B, *C,     ACQUIRE, *D,            RELEASE, *E, *F
        *A, *B,         ACQUIRE, *C,            RELEASE, *D, *E, *F
        *B,             ACQUIRE, *C, *D,        RELEASE, {*F,*A}, *E



INTERRUPT DISABLING FUNCTIONS
-----------------------------

Functions that disable interrupts (ACQUIRE equivalent) and enable interrupts
(RELEASE equivalent) will act as compiler barriers only.  So if memory or I/O
barriers are required in such a situation, they must be provided from some
other means.


SLEEP AND WAKE-UP FUNCTIONS
---------------------------

Sleeping and waking on an event flagged in global data can be viewed as an
interaction between two pieces of data: the task state of the task waiting for
the event and the global data used to indicate the event.  To make sure that
these appear to happen in the right order, the primitives to begin the process
of going to sleep, and the primitives to initiate a wake up imply certain
barriers.

Firstly, the sleeper normally follows something like this sequence of events:

        for (;;) {
                set_current_state(TASK_UNINTERRUPTIBLE);
                if (event_indicated)
                        break;
                schedule();
```

```
        }
```

A general memory barrier is interpolated automatically by set_current_state() after it has altered the task state:

```
        CPU 1
        ===============================
        set_current_state();
          smp_store_mb();
            STORE current->state
            <general barrier>
        LOAD event_indicated
```

set_current_state() may be wrapped by:

```
        prepare_to_wait();
        prepare_to_wait_exclusive();
```

which therefore also imply a general memory barrier after setting the state. The whole sequence above is available in various canned forms, all of which interpolate the memory barrier in the right place:

```
        wait_event();
        wait_event_interruptible();
        wait_event_interruptible_exclusive();
        wait_event_interruptible_timeout();
        wait_event_killable();
        wait_event_timeout();
        wait_on_bit();
        wait_on_bit_lock();
```

Secondly, code that performs a wake up normally follows something like this:

```
        event_indicated = 1;
        wake_up(&event_wait_queue);
```

or:

```
        event_indicated = 1;
        wake_up_process(event_daemon);
```

A general memory barrier is executed by wake_up() if it wakes something up. If it doesn't wake anything up then a memory barrier may or may not be executed; you must not rely on it. The barrier occurs before the task state is accessed, in particular, it sits between the STORE to indicate the event and the STORE to set TASK_RUNNING:

```
        CPU 1 (Sleeper)                 CPU 2 (Waker)
        =============================== ===============================
        set_current_state();            STORE event_indicated
          smp_store_mb();               wake_up();
            STORE current->state          ...
            <general barrier>             <general barrier>
        LOAD event_indicated              if ((LOAD task->state) & TASK_NORMAL)
                                            STORE task->state
```

where "task" is the thread being woken up and it equals CPU 1's "current".

To repeat, a general memory barrier is guaranteed to be executed by wake_up() if something is actually awakened, but otherwise there is no such guarantee. To see this, consider the following sequence of events, where X and Y are both initially zero:

```
        CPU 1                           CPU 2
        =============================== ===============================
        X = 1;                          Y = 1;
        smp_mb();                       wake_up();
        LOAD Y                          LOAD X

If a wakeup does occur, one (at least) of the two loads must see 1.  If, on
the other hand, a wakeup does not occur, both loads might see 0.

wake_up_process() always executes a general memory barrier.  The barrier again
occurs before the task state is accessed.  In particular, if the wake_up() in
the previous snippet were replaced by a call to wake_up_process() then one of
the two loads would be guaranteed to see 1.

The available waker functions include:

        complete();
        wake_up();
        wake_up_all();
        wake_up_bit();
        wake_up_interruptible();
        wake_up_interruptible_all();
        wake_up_interruptible_nr();
        wake_up_interruptible_poll();
        wake_up_interruptible_sync();
        wake_up_interruptible_sync_poll();
        wake_up_locked();
        wake_up_locked_poll();
        wake_up_nr();
        wake_up_poll();
        wake_up_process();

In terms of memory ordering, these functions all provide the same guarantees of
a wake_up() (or stronger).

[!] Note that the memory barriers implied by the sleeper and the waker do _not_
order multiple stores before the wake-up with respect to loads of those stored
values after the sleeper has called set_current_state().  For instance, if the
sleeper does:

        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated)
                break;
        __set_current_state(TASK_RUNNING);
        do_something(my_data);

and the waker does:

        my_data = value;
        event_indicated = 1;
        wake_up(&event_wait_queue);

there's no guarantee that the change to event_indicated will be perceived by
the sleeper as coming after the change to my_data.  In such a circumstance, the
code on both sides must interpolate its own memory barriers between the
separate data accesses.  Thus the above sleeper ought to do:

        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated) {
                smp_rmb();
                do_something(my_data);
        }
```

```
and the waker should do:

        my_data = value;
        smp_wmb();
        event_indicated = 1;
        wake_up(&event_wait_queue);


MISCELLANEOUS FUNCTIONS
-----------------------

Other functions that imply barriers:

 (*) schedule() and similar imply full memory barriers.


===================================
INTER-CPU ACQUIRING BARRIER EFFECTS
===================================

On SMP systems locking primitives give a more substantial form of barrier: one
that does affect memory access ordering on other CPUs, within the context of
conflict on any particular lock.


ACQUIRES VS MEMORY ACCESSES
---------------------------

Consider the following: the system has a pair of spinlocks (M) and (Q), and
three CPUs; then should the following sequence of events occur:

        CPU 1                           CPU 2
        =============================== ===============================
        WRITE_ONCE(*A, a);              WRITE_ONCE(*E, e);
        ACQUIRE M                       ACQUIRE Q
        WRITE_ONCE(*B, b);              WRITE_ONCE(*F, f);
        WRITE_ONCE(*C, c);              WRITE_ONCE(*G, g);
        RELEASE M                       RELEASE Q
        WRITE_ONCE(*D, d);              WRITE_ONCE(*H, h);

Then there is no guarantee as to what order CPU 3 will see the accesses to *A
through *H occur in, other than the constraints imposed by the separate locks
on the separate CPUs.  It might, for example, see:

        *E, ACQUIRE M, ACQUIRE Q, *G, *C, *F, *A, *B, RELEASE Q, *D, *H, RELEASE M

But it won't see any of:

        *B, *C or *D preceding ACQUIRE M
        *A, *B or *C following RELEASE M
        *F, *G or *H preceding ACQUIRE Q
        *E, *F or *G following RELEASE Q


=================================
WHERE ARE MEMORY BARRIERS NEEDED?
=================================

Under normal operation, memory operation reordering is generally not going to
be a problem as a single-threaded linear piece of code will still appear to
work correctly, even if it's in an SMP kernel.  There are, however, four
circumstances in which reordering definitely _could_ be a problem:
```

```
 (*) Interprocessor interaction.

 (*) Atomic operations.

 (*) Accessing devices.

 (*) Interrupts.


INTERPROCESSOR INTERACTION
--------------------------

When there's a system with more than one processor, more than one CPU in the
system may be working on the same data set at the same time.  This can cause
synchronisation problems, and the usual way of dealing with them is to use
locks.  Locks, however, are quite expensive, and so it may be preferable to
operate without the use of a lock if at all possible.  In such a case
operations that affect both CPUs may have to be carefully ordered to prevent
a malfunction.

Consider, for example, the R/W semaphore slow path.  Here a waiting process is
queued on the semaphore, by virtue of it having a piece of its stack linked to
the semaphore's list of waiting processes:

        struct rw_semaphore {
                ...
                spinlock_t lock;
                struct list_head waiters;
        };

        struct rwsem_waiter {
                struct list_head list;
                struct task_struct *task;
        };

To wake up a particular waiter, the up_read() or up_write() functions have to:

 (1) read the next pointer from this waiter's record to know as to where the
     next waiter record is;

 (2) read the pointer to the waiter's task structure;

 (3) clear the task pointer to tell the waiter it has been given the semaphore;

 (4) call wake_up_process() on the task; and

 (5) release the reference held on the waiter's task struct.

In other words, it has to perform this sequence of events:

        LOAD waiter->list.next;
        LOAD waiter->task;
        STORE waiter->task;
        CALL wakeup
        RELEASE task

and if any of these steps occur out of order, then the whole thing may
malfunction.

Once it has queued itself and dropped the semaphore lock, the waiter does not
get the lock again; it instead just waits for its task pointer to be cleared
before proceeding.  Since the record is on the waiter's stack, this means that
```

```
if the task pointer is cleared _before_ the next pointer in the list is read,
another CPU might start processing the waiter and might clobber the waiter's
stack before the up*() function has a chance to read the next pointer.

Consider then what might happen to the above sequence of events:

        CPU 1                           CPU 2
        =============================== ===============================
                                        down_xxx()
                                        Queue waiter
                                        Sleep
        up_yyy()
        LOAD waiter->task;
        STORE waiter->task;
                                        Woken up by other event
        <preempt>
                                        Resume processing
                                        down_xxx() returns
                                        call foo()
                                        foo() clobbers *waiter
        </preempt>
        LOAD waiter->list.next;
        --- OOPS ---

This could be dealt with using the semaphore lock, but then the down_xxx()
function has to needlessly get the spinlock again after being woken up.

The way to deal with this is to insert a general SMP memory barrier:

        LOAD waiter->list.next;
        LOAD waiter->task;
        smp_mb();
        STORE waiter->task;
        CALL wakeup
        RELEASE task

In this case, the barrier makes a guarantee that all memory accesses before the
barrier will appear to happen before all the memory accesses after the barrier
with respect to the other CPUs on the system.  It does _not_ guarantee that all
the memory accesses before the barrier will be complete by the time the barrier
instruction itself is complete.

On a UP system - where this wouldn't be a problem - the smp_mb() is just a
compiler barrier, thus making sure the compiler emits the instructions in the
right order without actually intervening in the CPU.  Since there's only one
CPU, that CPU's dependency ordering logic will take care of everything else.


ATOMIC OPERATIONS
-----------------

While they are technically interprocessor interaction considerations, atomic
operations are noted specially as some of them imply full memory barriers and
some don't, but they're very heavily relied on as a group throughout the
kernel.

See Documentation/atomic_t.txt for more information.


ACCESSING DEVICES
-----------------

Many devices can be memory mapped, and so appear to the CPU as if they're just
```

a set of memory locations. To control such a device, the driver usually has to make the right memory accesses in exactly the right order.

However, having a clever CPU or a clever compiler creates a potential problem in that the carefully sequenced accesses in the driver code won't reach the device in the requisite order if the CPU or the compiler thinks it is more efficient to reorder, combine or merge accesses - something that would cause the device to malfunction.

Inside of the Linux kernel, I/O should be done through the appropriate accessor routines - such as inb() or writel() - which know how to make such accesses appropriately sequential. While this, for the most part, renders the explicit use of memory barriers unnecessary, if the accessor functions are used to refer to an I/O memory window with relaxed memory access properties, then _mandatory_ memory barriers are required to enforce ordering.

See Documentation/driver-api/device-io.rst for more information.

INTERRUPTS

A driver may be interrupted by its own interrupt service routine, and thus the two parts of the driver may interfere with each other's attempts to control or access the device.

This may be alleviated - at least in part - by disabling local interrupts (a form of locking), such that the critical operations are all contained within the interrupt-disabled section in the driver. While the driver's interrupt routine is executing, the driver's core may not run on the same CPU, and its interrupt is not permitted to happen again until the current interrupt has been handled, thus the interrupt handler does not need to lock against that.

However, consider a driver that was talking to an ethernet card that sports an address register and a data register. If that driver's core talks to the card under interrupt-disablement and then the driver's interrupt handler is invoked:

```
        LOCAL IRQ DISABLE
        writew(ADDR, 3);
        writew(DATA, y);
        LOCAL IRQ ENABLE
        <interrupt>
        writew(ADDR, 4);
        q = readw(DATA);
        </interrupt>
```

The store to the data register might happen after the second store to the address register if ordering rules are sufficiently relaxed:

```
        STORE *ADDR = 3, STORE *ADDR = 4, STORE *DATA = y, q = LOAD *DATA
```

If ordering rules are relaxed, it must be assumed that accesses done inside an interrupt disabled section may leak outside of it and may interleave with accesses performed in an interrupt - and vice versa - unless implicit or explicit barriers are used.

Normally this won't be a problem because the I/O accesses done inside such sections will include synchronous load operations on strictly ordered I/O registers that form implicit I/O barriers.

A similar situation may occur between an interrupt routine and two routines

```
running on separate CPUs that communicate with each other.  If such a case is
likely, then interrupt-disabling locks should be used to guarantee ordering.


==========================
KERNEL I/O BARRIER EFFECTS
==========================

Interfacing with peripherals via I/O accesses is deeply architecture and device
specific. Therefore, drivers which are inherently non-portable may rely on
specific behaviours of their target systems in order to achieve synchronization
in the most lightweight manner possible. For drivers intending to be portable
between multiple architectures and bus implementations, the kernel offers a
series of accessor functions that provide various degrees of ordering
guarantees:

 (*) readX(), writeX():

        The readX() and writeX() MMIO accessors take a pointer to the
        peripheral being accessed as an __iomem * parameter. For pointers
        mapped with the default I/O attributes (e.g. those returned by
        ioremap()), the ordering guarantees are as follows:

        1. All readX() and writeX() accesses to the same peripheral are ordered
           with respect to each other. This ensures that MMIO register accesses
           by the same CPU thread to a particular device will arrive in program
           order.

        2. A writeX() issued by a CPU thread holding a spinlock is ordered
           before a writeX() to the same peripheral from another CPU thread
           issued after a later acquisition of the same spinlock. This ensures
           that MMIO register writes to a particular device issued while holding
           a spinlock will arrive in an order consistent with acquisitions of
           the lock.

        3. A writeX() by a CPU thread to the peripheral will first wait for the
           completion of all prior writes to memory either issued by, or
           propagated to, the same thread. This ensures that writes by the CPU
           to an outbound DMA buffer allocated by dma_alloc_coherent() will be
           visible to a DMA engine when the CPU writes to its MMIO control
           register to trigger the transfer.

        4. A readX() by a CPU thread from the peripheral will complete before
           any subsequent reads from memory by the same thread can begin. This
           ensures that reads by the CPU from an incoming DMA buffer allocated
           by dma_alloc_coherent() will not see stale data after reading from
           the DMA engine's MMIO status register to establish that the DMA
           transfer has completed.

        5. A readX() by a CPU thread from the peripheral will complete before
           any subsequent delay() loop can begin execution on the same thread.
           This ensures that two MMIO register writes by the CPU to a peripheral
           will arrive at least 1us apart if the first write is immediately read
           back with readX() and udelay(1) is called prior to the second
           writeX():

                writel(42, DEVICE_REGISTER_0); // Arrives at the device...
                readl(DEVICE_REGISTER_0);
                udelay(1);
                writel(42, DEVICE_REGISTER_1); // ...at least 1us before this.

        The ordering properties of __iomem pointers obtained with non-default
        attributes (e.g. those returned by ioremap_wc()) are specific to the
```

```
        underlying architecture and therefore the guarantees listed above cannot
        generally be relied upon for accesses to these types of mappings.

 (*) readX_relaxed(), writeX_relaxed():

        These are similar to readX() and writeX(), but provide weaker memory
        ordering guarantees. Specifically, they do not guarantee ordering with
        respect to locking, normal memory accesses or delay() loops (i.e.
        bullets 2-5 above) but they are still guaranteed to be ordered with
        respect to other accesses from the same CPU thread to the same
        peripheral when operating on __iomem pointers mapped with the default
        I/O attributes.

 (*) readsX(), writesX():

        The readsX() and writesX() MMIO accessors are designed for accessing
        register-based, memory-mapped FIFOs residing on peripherals that are not
        capable of performing DMA. Consequently, they provide only the ordering
        guarantees of readX_relaxed() and writeX_relaxed(), as documented above.

 (*) inX(), outX():

        The inX() and outX() accessors are intended to access legacy port-mapped
        I/O peripherals, which may require special instructions on some
        architectures (notably x86). The port number of the peripheral being
        accessed is passed as an argument.

        Since many CPU architectures ultimately access these peripherals via an
        internal virtual memory mapping, the portable ordering guarantees
        provided by inX() and outX() are the same as those provided by readX()
        and writeX() respectively when accessing a mapping with the default I/O
        attributes.

        Device drivers may expect outX() to emit a non-posted write transaction
        that waits for a completion response from the I/O peripheral before
        returning. This is not guaranteed by all architectures and is therefore
        not part of the portable ordering semantics.

 (*) insX(), outsX():

        As above, the insX() and outsX() accessors provide the same ordering
        guarantees as readsX() and writesX() respectively when accessing a
        mapping with the default I/O attributes.

 (*) ioreadX(), iowriteX():

        These will perform appropriately for the type of access they're actually
        doing, be it inX()/outX() or readX()/writeX().

With the exception of the string accessors (insX(), outsX(), readsX() and
writesX()), all of the above assume that the underlying peripheral is
little-endian and will therefore perform byte-swapping operations on big-endian
architectures.


========================================
ASSUMED MINIMUM EXECUTION ORDERING MODEL
========================================

It has to be assumed that the conceptual CPU is weakly-ordered but that it will
maintain the appearance of program causality with respect to itself.  Some CPUs
(such as i386 or x86_64) are more constrained than others (such as powerpc or
frv), and so the most relaxed case (namely DEC Alpha) must be assumed outside
```

```
of arch-specific code.

This means that it must be considered that the CPU will execute its instruction
stream in any order it feels like - or even in parallel - provided that if an
instruction in the stream depends on an earlier instruction, then that
earlier instruction must be sufficiently complete[*] before the later
instruction may proceed; in other words: provided that the appearance of
causality is maintained.

 [*] Some instructions have more than one effect - such as changing the
     condition codes, changing registers or changing memory - and different
     instructions may depend on different effects.

A CPU may also discard any instruction sequence that winds up having no
ultimate effect.  For example, if two adjacent instructions both load an
immediate value into the same register, the first may be discarded.


Similarly, it has to be assumed that compiler might reorder the instruction
stream in any way it sees fit, again provided the appearance of causality is
maintained.


============================
THE EFFECTS OF THE CPU CACHE
============================

The way cached memory operations are perceived across the system is affected to
a certain extent by the caches that lie between CPUs and memory, and by the
memory coherence system that maintains the consistency of state in the system.

As far as the way a CPU interacts with another part of the system through the
caches goes, the memory system has to include the CPU's caches, and memory
barriers for the most part act at the interface between the CPU and its cache
(memory barriers logically act on the dotted line in the following diagram):

            <--- CPU --->         :       <----------- Memory ----------->
                                  :
        +--------+    +--------+  :   +--------+    +-----------+
        |        |    |        |  :   |        |    |           |    +--------+
        |  CPU   |    | Memory |  :   | CPU    |    |           |    |        |
        |  Core  |--->| Access |----->| Cache  |<-->|           |    |        |
        |        |    | Queue  |  :   |        |    |           |--->| Memory |
        |        |    |        |  :   |        |    |           |    |        |
        +--------+    +--------+  :   +--------+    |           |    |        |
                                  :                 | Cache     |    +--------+
                                  :                 | Coherency |
                                  :                 | Mechanism |    +--------+
        +--------+    +--------+  :   +--------+    |           |    |        |
        |        |    |        |  :   |        |    |           |    |        |
        |  CPU   |    | Memory |  :   | CPU    |    |           |--->| Device |
        |  Core  |--->| Access |----->| Cache  |<-->|           |    |        |
        |        |    | Queue  |  :   |        |    |           |    |        |
        |        |    |        |  :   |        |    |           |    +--------+
        +--------+    +--------+  :   +--------+    +-----------+
                                  :
                                  :

Although any particular load or store may not actually appear outside of the
CPU that issued it since it may have been satisfied within the CPU's own cache,
it will still appear as if the full memory access had taken place as far as the
other CPUs are concerned since the cache coherency mechanisms will migrate the
cacheline over to the accessing CPU and propagate the effects upon conflict.
```

```
The CPU core may execute instructions in any order it deems fit, provided the
expected program causality appears to be maintained.  Some of the instructions
generate load and store operations which then go into the queue of memory
accesses to be performed.  The core may place these in the queue in any order
it wishes, and continue execution until it is forced to wait for an instruction
to complete.

What memory barriers are concerned with is controlling the order in which
accesses cross from the CPU side of things to the memory side of things, and
the order in which the effects are perceived to happen by the other observers
in the system.

[!] Memory barriers are _not_ needed within a given CPU, as CPUs always see
their own loads and stores as if they had happened in program order.

[!] MMIO or other device accesses may bypass the cache system.  This depends on
the properties of the memory window through which devices are accessed and/or
the use of any special device communication instructions the CPU may have.


CACHE COHERENCY VS DMA
----------------------

Not all systems maintain cache coherency with respect to devices doing DMA.  In
such cases, a device attempting DMA may obtain stale data from RAM because
dirty cache lines may be resident in the caches of various CPUs, and may not
have been written back to RAM yet.  To deal with this, the appropriate part of
the kernel must flush the overlapping bits of cache on each CPU (and maybe
invalidate them as well).

In addition, the data DMA'd to RAM by a device may be overwritten by dirty
cache lines being written back to RAM from a CPU's cache after the device has
installed its own data, or cache lines present in the CPU's cache may simply
obscure the fact that RAM has been updated, until at such time as the cacheline
is discarded from the CPU's cache and reloaded.  To deal with this, the
appropriate part of the kernel must invalidate the overlapping bits of the
cache on each CPU.

See Documentation/core-api/cachetlb.rst for more information on cache
management.


CACHE COHERENCY VS MMIO
-----------------------

Memory mapped I/O usually takes place through memory locations that are part of
a window in the CPU's memory space that has different properties assigned than
the usual RAM directed window.

Amongst these properties is usually the fact that such accesses bypass the
caching entirely and go directly to the device buses.  This means MMIO accesses
may, in effect, overtake accesses to cached memory that were emitted earlier.
A memory barrier isn't sufficient in such a case, but rather the cache must be
flushed between the cached memory write and the MMIO access if the two are in
any way dependent.


=========================
THE THINGS CPUS GET UP TO
=========================

A programmer might take it for granted that the CPU will perform memory
```

```
operations in exactly the order specified, so that if the CPU is, for example,
given the following piece of code to execute:

        a = READ_ONCE(*A);
        WRITE_ONCE(*B, b);
        c = READ_ONCE(*C);
        d = READ_ONCE(*D);
        WRITE_ONCE(*E, e);

they would then expect that the CPU will complete the memory operation for each
instruction before moving on to the next one, leading to a definite sequence of
operations as seen by external observers in the system:

        LOAD *A, STORE *B, LOAD *C, LOAD *D, STORE *E.


Reality is, of course, much messier.  With many CPUs and compilers, the above
assumption doesn't hold because:

 (*) loads are more likely to need to be completed immediately to permit
     execution progress, whereas stores can often be deferred without a
     problem;

 (*) loads may be done speculatively, and the result discarded should it prove
     to have been unnecessary;

 (*) loads may be done speculatively, leading to the result having been fetched
     at the wrong time in the expected sequence of events;

 (*) the order of the memory accesses may be rearranged to promote better use
     of the CPU buses and caches;

 (*) loads and stores may be combined to improve performance when talking to
     memory or I/O hardware that can do batched accesses of adjacent locations,
     thus cutting down on transaction setup costs (memory and PCI devices may
     both be able to do this); and

 (*) the CPU's data cache may affect the ordering, and while cache-coherency
     mechanisms may alleviate this - once the store has actually hit the cache
     - there's no guarantee that the coherency management will be propagated in
     order to other CPUs.

So what another CPU, say, might actually observe from the above piece of code
is:

        LOAD *A, ..., LOAD {*C,*D}, STORE *E, STORE *B

        (Where "LOAD {*C,*D}" is a combined load)


However, it is guaranteed that a CPU will be self-consistent: it will see its
_own_ accesses appear to be correctly ordered, without the need for a memory
barrier.  For instance with the following code:

        U = READ_ONCE(*A);
        WRITE_ONCE(*A, V);
        WRITE_ONCE(*A, W);
        X = READ_ONCE(*A);
        WRITE_ONCE(*A, Y);
        Z = READ_ONCE(*A);

and assuming no intervention by an external influence, it can be assumed that
the final result will appear to be:
```

```
        U == the original value of *A
        X == W
        Z == Y
        *A == Y
```

The code above may cause the CPU to generate the full sequence of memory accesses:

```
        U=LOAD *A, STORE *A=V, STORE *A=W, X=LOAD *A, STORE *A=Y, Z=LOAD *A
```

in that order, but, without intervention, the sequence may have almost any combination of elements combined or discarded, provided the program's view of the world remains consistent. Note that READ_ONCE() and WRITE_ONCE() are -not- optional in the above example, as there are architectures where a given CPU might reorder successive loads to the same location. On such architectures, READ_ONCE() and WRITE_ONCE() do whatever is necessary to prevent this, for example, on Itanium the volatile casts used by READ_ONCE() and WRITE_ONCE() cause GCC to emit the special ld.acq and st.rel instructions (respectively) that prevent such reordering.

The compiler may also combine, discard or defer elements of the sequence before the CPU even sees them.

For instance:

```
        *A = V;
        *A = W;
```

may be reduced to:

```
        *A = W;
```

since, without either a write barrier or an WRITE_ONCE(), it can be assumed that the effect of the storage of V to *A is lost. Similarly:

```
        *A = Y;
        Z = *A;
```

may, without a memory barrier or an READ_ONCE() and WRITE_ONCE(), be reduced to:

```
        *A = Y;
        Z = Y;
```

and the LOAD operation never appear outside of the CPU.

AND THEN THERE'S THE ALPHA

The DEC Alpha CPU is one of the most relaxed CPUs there is. Not only that, some versions of the Alpha CPU have a split data cache, permitting them to have two semantically-related cache lines updated at separate times. This is where the address-dependency barrier really becomes necessary as this synchronises both caches with the memory coherence system, thus making it seem like pointer changes vs new data occur in the right order.

The Alpha defines the Linux kernel's memory model, although as of v4.15 the Linux kernel's addition of smp_mb() to READ_ONCE() on Alpha greatly reduced its impact on the memory model.

```
VIRTUAL MACHINE GUESTS
----------------------

Guests running within virtual machines might be affected by SMP effects even if
the guest itself is compiled without SMP support.  This is an artifact of
interfacing with an SMP host while running an UP kernel.  Using mandatory
barriers for this use-case would be possible but is often suboptimal.

To handle this case optimally, low-level virt_mb() etc macros are available.
These have the same effect as smp_mb() etc when SMP is enabled, but generate
identical code for SMP and non-SMP systems.  For example, virtual machine guests
should use virt_mb() rather than smp_mb() when synchronizing against a
(possibly SMP) host.

These are equivalent to smp_mb() etc counterparts in all other respects,
in particular, they do not control MMIO effects: to control
MMIO effects, use mandatory barriers.


============
EXAMPLE USES
============

CIRCULAR BUFFERS
----------------

Memory barriers can be used to implement circular buffering without the need
of a lock to serialise the producer with the consumer.  See:

        Documentation/core-api/circular-buffers.rst

for details.


==========
REFERENCES
==========

Alpha AXP Architecture Reference Manual, Second Edition (Sites & Witek,
Digital Press)
        Chapter 5.2: Physical Address Space Characteristics
        Chapter 5.4: Caches and Write Buffers
        Chapter 5.5: Data Sharing
        Chapter 5.6: Read/Write Ordering

AMD64 Architecture Programmer's Manual Volume 2: System Programming
        Chapter 7.1: Memory-Access Ordering
        Chapter 7.4: Buffering and Combining Memory Writes

ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)
        Chapter B2: The AArch64 Application Level Memory Model

IA-32 Intel Architecture Software Developer's Manual, Volume 3:
System Programming Guide
        Chapter 7.1: Locked Atomic Operations
        Chapter 7.2: Memory Ordering
        Chapter 7.4: Serializing Instructions

The SPARC Architecture Manual, Version 9
        Chapter 8: Memory Models
        Appendix D: Formal Specification of the Memory Models
        Appendix J: Programming with the Memory Models
```

```
Storage in the PowerPC (Stone and Fitzgerald)

UltraSPARC Programmer Reference Manual
        Chapter 5: Memory Accesses and Cacheability
        Chapter 15: Sparc-V9 Memory Models

UltraSPARC III Cu User's Manual
        Chapter 9: Memory Models

UltraSPARC IIIi Processor User's Manual
        Chapter 8: Memory Models

UltraSPARC Architecture 2005
        Chapter 9: Memory
        Appendix D: Formal Specifications of the Memory Models

UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005
        Chapter 8: Memory Models
        Appendix F: Caches and Cache Coherency

Solaris Internals, Core Kernel Architecture, p63-68:
        Chapter 3.3: Hardware Considerations for Locks and
                        Synchronization

Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching
for Kernel Programmers:
        Chapter 13: Other Memory Models

Intel Itanium Architecture Software Developer's Manual: Volume 1:
        Section 2.6: Speculation
        Section 4.4: Memory Access
```

CHAPTER

FIVE

LOW-LEVEL HARDWARE MANAGEMENT

Cache management, managing CPU hotplug, etc.

5.1 Cache and TLB Flushing Under Linux

Author
David S. Miller <davem@redhat.com>

This document describes the cache/tlb flushing interfaces called by the Linux VM subsystem. It enumerates over each interface, describes its intended purpose, and what side effect is expected after the interface is invoked.

The side effects described below are stated for a uniprocessor implementation, and what is to happen on that single processor. The SMP cases are a simple extension, in that you just extend the definition such that the side effect for a particular interface occurs on all processors in the system. Don't let this scare you into thinking SMP cache/tlb flushing must be so inefficient, this is in fact an area where many optimizations are possible. For example, if it can be proven that a user address space has never executed on a cpu (see mm_cpumask()), one need not perform a flush for this address space on that cpu.

First, the TLB flushing interfaces, since they are the simplest. The "TLB" is abstracted under Linux as something the cpu uses to cache virtual-->physical address translations obtained from the software page tables. Meaning that if the software page tables change, it is possible for stale translations to exist in this "TLB" cache. Therefore when software page table changes occur, the kernel will invoke one of the following flush methods _after_ the page table changes occur:

1) `void flush_tlb_all(void)`

   The most severe flush of all. After this interface runs, any previous page table modification whatsoever will be visible to the cpu.

   This is usually invoked when the kernel page tables are changed, since such translations are "global" in nature.

2) `void flush_tlb_mm(struct mm_struct *mm)`

   This interface flushes an entire user address space from the TLB. After running, this interface must make sure that any previous page table modifications for the address space 'mm' will be visible to the cpu. That is, after running, there will be no entries in the TLB for 'mm'.

This interface is used to handle whole address space page table operations such as what happens during fork, and exec.

3) `void flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)`

   Here we are flushing a specific range of (user) virtual address translations from the TLB. After running, this interface must make sure that any previous page table modifications for the address space 'vma->vm_mm' in the range 'start' to 'end-1' will be visible to the cpu. That is, after running, there will be no entries in the TLB for 'mm' for virtual addresses in the range 'start' to 'end-1'.

   The "vma" is the backing store being used for the region. Primarily, this is used for munmap() type operations.

   The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized translations from the TLB, instead of having the kernel call flush_tlb_page (see below) for each entry which may be modified.

4) `void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)`

   This time we need to remove the PAGE_SIZE sized translation from the TLB. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process, the address space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if this region is executable (and thus could be in the 'instruction TLB' in split-tlb type setups).

   After running, this interface must make sure that any previous page table modification for address space 'vma->vm_mm' for user virtual address 'addr' will be visible to the cpu. That is, after running, there will be no entries in the TLB for 'vma->vm_mm' for virtual address 'addr'.

   This is used primarily during fault processing.

5) `void update_mmu_cache_range(struct vm_fault *vmf, struct vm_area_struct *vma, unsigned long address, pte_t *ptep, unsigned int nr)`

   At the end of every page fault, this routine is invoked to tell the architecture specific code that translations now exists in the software page tables for address space "vma->vm_mm" at virtual address "address" for "nr" consecutive pages.

   This routine is also invoked in various other places which pass a NULL "vmf".

   A port may use this information in any way it so chooses. For example, it could use this event to pre-load TLB translations for software managed TLB configurations. The sparc64 port currently does this.

Next, we have the cache flushing interfaces. In general, when Linux is changing an existing virtual-->physical mapping to a new value, the sequence will be in one of the following forms:

```
1) flush_cache_mm(mm);
   change_all_page_tables_of(mm);
   flush_tlb_mm(mm);

2) flush_cache_range(vma, start, end);
   change_range_of_page_tables(mm, start, end);
   flush_tlb_range(vma, start, end);
```

```
3) flush_cache_page(vma, addr, pfn);
   set_pte(pte_pointer, new_pte_val);
   flush_tlb_page(vma, addr);
```

The cache level flush will always be first, because this allows us to properly handle systems whose caches are strict and require a virtual-->physical translation to exist for a virtual address when that virtual address is flushed from the cache. The HyperSparc cpu is one such cpu with this attribute.

The cache flushing routines below need only deal with cache flushing to the extent that it is necessary for a particular cpu. Mostly, these routines must be implemented for cpus which have virtually indexed caches which must be flushed when virtual-->physical translations are changed or removed. So, for example, the physically indexed physically tagged caches of IA32 processors have no need to implement these interfaces since the caches are fully synchronized and have no dependency on translation information.

Here are the routines, one by one:

1) `void flush_cache_mm(struct mm_struct *mm)`

    This interface flushes an entire user address space from the caches. That is, after running, there will be no cache lines associated with 'mm'.

    This interface is used to handle whole address space page table operations such as what happens during exit and exec.

2) `void flush_cache_dup_mm(struct mm_struct *mm)`

    This interface flushes an entire user address space from the caches. That is, after running, there will be no cache lines associated with 'mm'.

    This interface is used to handle whole address space page table operations such as what happens during fork.

    This option is separate from flush_cache_mm to allow some optimizations for VIPT caches.

3) `void flush_cache_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)`

    Here we are flushing a specific range of (user) virtual addresses from the cache. After running, there will be no entries in the cache for 'vma->vm_mm' for virtual addresses in the range 'start' to 'end-1'.

    The "vma" is the backing store being used for the region. Primarily, this is used for munmap() type operations.

    The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized regions from the cache, instead of having the kernel call flush_cache_page (see below) for each entry which may be modified.

4) `void flush_cache_page(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn)`

    This time we need to remove a PAGE_SIZE sized range from the cache. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process, the address space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if this region is executable (and thus could be in the 'instruction cache' in "Harvard" type cache layouts).

The 'pfn' indicates the physical page frame (shift this value left by PAGE_SHIFT to get the physical address) that 'addr' translates to. It is this mapping which should be removed from the cache.

After running, there will be no entries in the cache for 'vma->vm_mm' for virtual address 'addr' which translates to 'pfn'.

This is used primarily during fault processing.

5) `void flush_cache_kmaps(void)`

   This routine need only be implemented if the platform utilizes highmem. It will be called right before all of the kmaps are invalidated.

   After running, there will be no entries in the cache for the kernel virtual address range PKMAP_ADDR(0) to PKMAP_ADDR(LAST_PKMAP).

   This routing should be implemented in asm/highmem.h

6) `void flush_cache_vmap(unsigned long start, unsigned long end)` `void flush_cache_vunmap(unsigned long start, unsigned long end)`

   Here in these two interfaces we are flushing a specific range of (kernel) virtual addresses from the cache. After running, there will be no entries in the cache for the kernel address space for virtual addresses in the range 'start' to 'end-1'.

   The first of these two routines is invoked after vmap_range() has installed the page table entries. The second is invoked before vunmap_range() deletes the page table entries.

There exists another whole class of cpu cache issues which currently require a whole different set of interfaces to handle properly. The biggest problem is that of virtual aliasing in the data cache of a processor.

Is your port susceptible to virtual aliasing in its D-cache? Well, if your D-cache is virtually indexed, is larger in size than PAGE_SIZE, and does not prevent multiple cache lines for the same physical address from existing at once, you have this problem.

If your D-cache has this problem, first define asm/shmparam.h SHMLBA properly, it should essentially be the size of your virtually addressed D-cache (or if the size is variable, the largest possible size). This setting will force the SYSv IPC layer to only allow user processes to mmap shared memory at address which are a multiple of this value.

Note: This does not fix shared mmaps, check out the sparc64 port for one way to solve this (in particular SPARC_FLAG_MMAPSHARED).

Next, you have to solve the D-cache aliasing issue for all other cases. Please keep in mind that fact that, for a given page mapped into some user address space, there is always at least one more mapping, that of the kernel in its linear mapping starting at PAGE_OFFSET. So immediately, once the first user maps a given physical page into its address space, by implication the D-cache aliasing problem has the potential to exist since the kernel already maps this page at its virtual address.

`void copy_user_page(void *to, void *from, unsigned long addr, struct page *page)` `void clear_user_page(void *to, unsigned long addr, struct page *page)`

These two routines store data in user anonymous or COW pages. It allows a port to efficiently avoid D-cache alias issues between userspace and the kernel.

For example, a port may temporarily map 'from' and 'to' to kernel virtual addresses during the copy. The virtual address for these two pages is chosen in such a way that the kernel load/store instructions happen to virtual addresses which are of the same "color" as the user mapping of the page. Sparc64 for example, uses this technique.

The 'addr' parameter tells the virtual address where the user will ultimately have this page mapped, and the 'page' parameter gives a pointer to the struct page of the target.

If D-cache aliasing is not an issue, these two routines may simply call memcpy/memset directly and do nothing more.

`void flush_dcache_folio(struct folio *folio)`

This routines must be called when:

a) the kernel did write to a page that is in the page cache page and / or in high memory

b) the kernel is about to read from a page cache page and user space shared/writable mappings of this page potentially exist. Note that {get,pin}_user_pages{_fast} already call flush_dcache_folio on any page found in the user address space and thus driver code rarely needs to take this into account.

Note: This routine need only be called for page cache pages which can potentially ever be mapped into the address space of a user process. So for example, VFS layer code handling vfs symlinks in the page cache need not call this interface at all.

The phrase "kernel writes to a page cache page" means, specifically, that the kernel executes store instructions that dirty data in that page at the kernel virtual mapping of that page. It is important to flush here to handle D-cache aliasing, to make sure these kernel stores are visible to user space mappings of that page.

The corollary case is just as important, if there are users which have shared+writable mappings of this file, we must make sure that kernel reads of these pages will see the most recent stores done by the user.

If D-cache aliasing is not an issue, this routine may simply be defined as a nop on that architecture.

There is a bit set aside in folio->flags (PG_arch_1) as "architecture private". The kernel guarantees that, for pagecache pages, it will clear this bit when such a page first enters the pagecache.

This allows these interfaces to be implemented much more efficiently. It allows one to "defer" (perhaps indefinitely) the actual flush if there are currently no user processes mapping this page. See sparc64's flush_dcache_folio and update_mmu_cache_range implementations for an example of how to go about doing this.

The idea is, first at flush_dcache_folio() time, if *folio_flush_mapping()* returns a mapping, and mapping_mapped() on that mapping returns %false, just mark the architecture private page flag bit. Later, in update_mmu_cache_range(), a check is made of this flag bit, and if set the flush is done and the flag bit is cleared.

Important: It is often important, if you defer the flush, that the actual flush occurs on the same CPU as did the cpu stores into the page to make it dirty. Again, see sparc64 for examples of how to deal with this.

`void copy_to_user_page(struct vm_area_struct *vma, struct page *page, unsigned long user_vaddr, void *dst, void *src, int len)` `void copy_from_user_page(struct vm_area_struct *vma, struct page *page, unsigned long user_vaddr, void *dst, void *src, int len)`

When the kernel needs to copy arbitrary data in and out of arbitrary user pages (f.e. for ptrace()) it will use these two routines.

Any necessary cache flushing or other coherency operations that need to occur should happen here. If the processor's instruction cache does not snoop cpu stores, it is very likely that you will need to flush the instruction cache for copy_to_user_page().

`void flush_anon_page(struct vm_area_struct *vma, struct page *page, unsigned long vmaddr)`

When the kernel needs to access the contents of an anonymous page, it calls this function (currently only get_user_pages()). Note: flush_dcache_folio() deliberately doesn't work for an anonymous page. The default implementation is a nop (and should remain so for all coherent architectures). For incoherent architectures, it should flush the cache of the page at vmaddr.

`void flush_icache_range(unsigned long start, unsigned long end)`

When the kernel stores into addresses that it will execute out of (eg when loading modules), this function is called.

If the icache does not snoop stores then this routine will need to flush it.

`void flush_icache_page(struct vm_area_struct *vma, struct page *page)`

All the functionality of flush_icache_page can be implemented in flush_dcache_folio and update_mmu_cache_range. In the future, the hope is to remove this interface completely.

The final category of APIs is for I/O to deliberately aliased address ranges inside the kernel. Such aliases are set up by use of the vmap/vmalloc API. Since kernel I/O goes via physical pages, the I/O subsystem assumes that the user mapping and kernel offset mapping are the only aliases. This isn't true for vmap aliases, so anything in the kernel trying to do I/O to vmap areas must manually manage coherency. It must do this by flushing the vmap range before doing I/O and invalidating it after the I/O returns.

`void flush_kernel_vmap_range(void *vaddr, int size)`

flushes the kernel cache for a given virtual address range in the vmap area. This is to make sure that any data the kernel modified in the vmap range is made visible to the physical page. The design is to make this area safe to perform I/O on. Note that this API does *not* also flush the offset map alias of the area.

`void invalidate_kernel_vmap_range(void *vaddr, int size) invalidates`

the cache for a given virtual address range in the vmap area which prevents the processor from making the cache stale by speculatively reading data while the I/O was occurring to the physical pages. This is only necessary for data reads into the vmap area.

5.2 CPU hotplug in the Kernel

Date
September, 2021

Author
Sebastian Andrzej Siewior <bigeasy@linutronix.de>, Rusty Russell <rusty@rustcorp.com.au>, Srivatsa Vaddagiri <vatsa@in.ibm.com>, Ashok Raj <ashok.raj@intel.com>, Joel Schopp <jschopp@austin.ibm.com>, Thomas Gleixner <tglx@linutronix.de>

5.2.1 Introduction

Modern advances in system architectures have introduced advanced error reporting and correction capabilities in processors. There are couple OEMS that support NUMA hardware which are hot pluggable as well, where physical node insertion and removal require support for CPU hotplug.

Such advances require CPUs available to a kernel to be removed either for provisioning reasons, or for RAS purposes to keep an offending CPU off system execution path. Hence the need for CPU hotplug support in the Linux kernel.

A more novel use of CPU-hotplug support is its use today in suspend resume support for SMP. Dual-core and HT support makes even a laptop run SMP kernels which didn't support these methods.

5.2.2 Command Line Switches

maxcpus=n
: Restrict boot time CPUs to *n*. Say if you have four CPUs, using `maxcpus=2` will only boot two. You can choose to bring the other CPUs later online.

nr_cpus=n
: Restrict the total amount of CPUs the kernel will support. If the number supplied here is lower than the number of physically available CPUs, then those CPUs can not be brought online later.

additional_cpus=n
: Use this to limit hotpluggable CPUs. This option sets `cpu_possible_mask = cpu_present_mask + additional_cpus`

  This option is limited to the IA64 architecture.

possible_cpus=n
: This option sets `possible_cpus` bits in `cpu_possible_mask`.

  This option is limited to the X86 and S390 architecture.

cpu0_hotplug
: Allow to shutdown CPU0.

  This option is limited to the X86 architecture.

5.2.3 CPU maps

cpu_possible_mask
: Bitmap of possible CPUs that can ever be available in the system. This is used to allocate some boot time memory for per_cpu variables that aren't designed to grow/shrink as CPUs are made available or removed. Once set during boot time discovery phase, the map is static, i.e no bits are added or removed anytime. Trimming it accurately for your system needs upfront can save some boot time memory.

cpu_online_mask
: Bitmap of all CPUs currently online. Its set in `__cpu_up()` after a CPU is available for kernel scheduling and ready to receive interrupts from devices. Its cleared when a CPU is brought down using `__cpu_disable()`, before which all OS services including interrupts are migrated to another target CPU.

cpu_present_mask
: Bitmap of CPUs currently present in the system. Not all of them may be online. When physical hotplug is processed by the relevant subsystem (e.g ACPI) can change and new bit either be added or removed from the map depending on the event is hot-add/hot-remove. There are currently no locking rules as of now. Typical usage is to init topology during boot, at which time hotplug is disabled.

You really don't need to manipulate any of the system CPU maps. They should be read-only for most use. When setting up per-cpu resources almost always use `cpu_possible_mask` or `for_each_possible_cpu()` to iterate. To macro `for_each_cpu()` can be used to iterate over a custom CPU mask.

Never use anything other than `cpumask_t` to represent bitmap of CPUs.

5.2.4 Using CPU hotplug

The kernel option *CONFIG_HOTPLUG_CPU* needs to be enabled. It is currently available on multiple architectures including ARM, MIPS, PowerPC and X86. The configuration is done via the sysfs interface:

```
$ ls -lh /sys/devices/system/cpu
total 0
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu0
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu1
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu2
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu3
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu4
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu5
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu6
drwxr-xr-x  9 root root    0 Dec 21 16:33 cpu7
drwxr-xr-x  2 root root    0 Dec 21 16:33 hotplug
-r--r--r--  1 root root 4.0K Dec 21 16:33 offline
-r--r--r--  1 root root 4.0K Dec 21 16:33 online
-r--r--r--  1 root root 4.0K Dec 21 16:33 possible
-r--r--r--  1 root root 4.0K Dec 21 16:33 present
```

The files *offline*, *online*, *possible*, *present* represent the CPU masks. Each CPU folder contains an *online* file which controls the logical on (1) and off (0) state. To logically shutdown CPU4:

```
$ echo 0 > /sys/devices/system/cpu/cpu4/online
 smpboot: CPU 4 is now offline
```

Once the CPU is shutdown, it will be removed from */proc/interrupts*, */proc/cpuinfo* and should also not be shown visible by the *top* command. To bring CPU4 back online:

```
$ echo 1 > /sys/devices/system/cpu/cpu4/online
smpboot: Booting Node 0 Processor 4 APIC 0x1
```

The CPU is usable again. This should work on all CPUs, but CPU0 is often special and excluded from CPU hotplug.

5.2.5 The CPU hotplug coordination

The offline case

Once a CPU has been logically shutdown the teardown callbacks of registered hotplug states will be invoked, starting with `CPUHP_ONLINE` and terminating at state `CPUHP_OFFLINE`. This includes:

- If tasks are frozen due to a suspend operation then *cpuhp_tasks_frozen* will be set to true.
- All processes are migrated away from this outgoing CPU to new CPUs. The new CPU is chosen from each process' current cpuset, which may be a subset of all online CPUs.
- All interrupts targeted to this CPU are migrated to a new CPU
- timers are also migrated to a new CPU

- Once all services are migrated, kernel calls an arch specific routine `__cpu_disable()` to perform arch specific cleanup.

5.2.6 The CPU hotplug API

CPU hotplug state machine

CPU hotplug uses a trivial state machine with a linear state space from CPUHP_OFFLINE to CPUHP_ONLINE. Each state has a startup and a teardown callback.

When a CPU is onlined, the startup callbacks are invoked sequentially until the state CPUHP_ONLINE is reached. They can also be invoked when the callbacks of a state are set up or an instance is added to a multi-instance state.

When a CPU is offlined the teardown callbacks are invoked in the reverse order sequentially until the state CPUHP_OFFLINE is reached. They can also be invoked when the callbacks of a state are removed or an instance is removed from a multi-instance state.

If a usage site requires only a callback in one direction of the hotplug operations (CPU online or CPU offline) then the other not-required callback can be set to NULL when the state is set up.

The state space is divided into three sections:

- The PREPARE section

  The PREPARE section covers the state space from CPUHP_OFFLINE to CPUHP_BRINGUP_CPU.

  The startup callbacks in this section are invoked before the CPU is started during a CPU online operation. The teardown callbacks are invoked after the CPU has become dysfunctional during a CPU offline operation.

  The callbacks are invoked on a control CPU as they can't obviously run on the hotplugged CPU which is either not yet started or has become dysfunctional already.

  The startup callbacks are used to setup resources which are required to bring a CPU successfully online. The teardown callbacks are used to free resources or to move pending work to an online CPU after the hotplugged CPU became dysfunctional.

  The startup callbacks are allowed to fail. If a callback fails, the CPU online operation is aborted and the CPU is brought down to the previous state (usually CPUHP_OFFLINE) again.

  The teardown callbacks in this section are not allowed to fail.

- The STARTING section

  The STARTING section covers the state space between CPUHP_BRINGUP_CPU + 1 and CPUHP_AP_ONLINE.

  The startup callbacks in this section are invoked on the hotplugged CPU with interrupts disabled during a CPU online operation in the early CPU setup code. The teardown callbacks are invoked with interrupts disabled on the hotplugged CPU during a CPU offline operation shortly before the CPU is completely shut down.

  The callbacks in this section are not allowed to fail.

The callbacks are used for low level hardware initialization/shutdown and for core subsystems.

- The ONLINE section

  The ONLINE section covers the state space between CPUHP_AP_ONLINE + 1 and CPUHP_ONLINE.

  The startup callbacks in this section are invoked on the hotplugged CPU during a CPU online operation. The teardown callbacks are invoked on the hotplugged CPU during a CPU offline operation.

  The callbacks are invoked in the context of the per CPU hotplug thread, which is pinned on the hotplugged CPU. The callbacks are invoked with interrupts and preemption enabled.

  The callbacks are allowed to fail. When a callback fails the hotplug operation is aborted and the CPU is brought back to the previous state.

CPU online/offline operations

A successful online operation looks like this:

```
[CPUHP_OFFLINE]
[CPUHP_OFFLINE + 1]->startup()       -> success
[CPUHP_OFFLINE + 2]->startup()       -> success
[CPUHP_OFFLINE + 3]                  -> skipped because startup == NULL
...
[CPUHP_BRINGUP_CPU]->startup()       -> success
=== End of PREPARE section
[CPUHP_BRINGUP_CPU + 1]->startup()   -> success
...
[CPUHP_AP_ONLINE]->startup()         -> success
=== End of STARTUP section
[CPUHP_AP_ONLINE + 1]->startup()     -> success
...
[CPUHP_ONLINE - 1]->startup()        -> success
[CPUHP_ONLINE]
```

A successful offline operation looks like this:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
[CPUHP_AP_ONLINE + 1]->teardown()    -> success
=== Start of STARTUP section
[CPUHP_AP_ONLINE]->teardown()        -> success
...
[CPUHP_BRINGUP_ONLINE - 1]->teardown()
...
=== Start of PREPARE section
[CPUHP_BRINGUP_CPU]->teardown()
[CPUHP_OFFLINE + 3]->teardown()
[CPUHP_OFFLINE + 2]                  -> skipped because teardown == NULL
```

```
[CPUHP_OFFLINE + 1]->teardown()
[CPUHP_OFFLINE]
```

A failed online operation looks like this:

```
[CPUHP_OFFLINE]
[CPUHP_OFFLINE + 1]->startup()       -> success
[CPUHP_OFFLINE + 2]->startup()       -> success
[CPUHP_OFFLINE + 3]                  -> skipped because startup == NULL
...
[CPUHP_BRINGUP_CPU]->startup()       -> success
=== End of PREPARE section
[CPUHP_BRINGUP_CPU + 1]->startup()   -> success
...
[CPUHP_AP_ONLINE]->startup()         -> success
=== End of STARTUP section
[CPUHP_AP_ONLINE + 1]->startup()     -> success
---
[CPUHP_AP_ONLINE + N]->startup()     -> fail
[CPUHP_AP_ONLINE + (N - 1)]->teardown()
...
[CPUHP_AP_ONLINE + 1]->teardown()
=== Start of STARTUP section
[CPUHP_AP_ONLINE]->teardown()
...
[CPUHP_BRINGUP_ONLINE - 1]->teardown()
...
=== Start of PREPARE section
[CPUHP_BRINGUP_CPU]->teardown()
[CPUHP_OFFLINE + 3]->teardown()
[CPUHP_OFFLINE + 2]                  -> skipped because teardown == NULL
[CPUHP_OFFLINE + 1]->teardown()
[CPUHP_OFFLINE]
```

A failed offline operation looks like this:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()
...
[CPUHP_ONLINE - 1]->startup()
[CPUHP_ONLINE]
```

Recursive failures cannot be handled sensibly. Look at the following example of a recursive fail due to a failed offline operation:

```
[CPUHP_ONLINE]
[CPUHP_ONLINE - 1]->teardown()       -> success
...
```

```
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()  -> success
[CPUHP_ONLINE - (N - 2)]->startup()  -> fail
```

The CPU hotplug state machine stops right here and does not try to go back down again because that would likely result in an endless loop:

```
[CPUHP_ONLINE - (N - 1)]->teardown() -> success
[CPUHP_ONLINE - N]->teardown()       -> fail
[CPUHP_ONLINE - (N - 1)]->startup()  -> success
[CPUHP_ONLINE - (N - 2)]->startup()  -> fail
[CPUHP_ONLINE - (N - 1)]->teardown() -> success
[CPUHP_ONLINE - N]->teardown()       -> fail
```

Lather, rinse and repeat. In this case the CPU left in state:

```
[CPUHP_ONLINE - (N - 1)]
```

which at least lets the system make progress and gives the user a chance to debug or even resolve the situation.

Allocating a state

There are two ways to allocate a CPU hotplug state:

- Static allocation

  Static allocation has to be used when the subsystem or driver has ordering requirements versus other CPU hotplug states. E.g. the PERF core startup callback has to be invoked before the PERF driver startup callbacks during a CPU online operation. During a CPU offline operation the driver teardown callbacks have to be invoked before the core teardown callback. The statically allocated states are described by constants in the cpuhp_state enum which can be found in include/linux/cpuhotplug.h.

  Insert the state into the enum at the proper place so the ordering requirements are fulfilled. The state constant has to be used for state setup and removal.

  Static allocation is also required when the state callbacks are not set up at runtime and are part of the initializer of the CPU hotplug state array in kernel/cpu.c.

- Dynamic allocation

  When there are no ordering requirements for the state callbacks then dynamic allocation is the preferred method. The state number is allocated by the setup function and returned to the caller on success.

  Only the PREPARE and ONLINE sections provide a dynamic allocation range. The STARTING section does not as most of the callbacks in that section have explicit ordering requirements.

Setup of a CPU hotplug state

The core code provides the following functions to setup a state:

- cpuhp_setup_state(state, name, startup, teardown)
- cpuhp_setup_state_nocalls(state, name, startup, teardown)
- cpuhp_setup_state_cpuslocked(state, name, startup, teardown)
- cpuhp_setup_state_nocalls_cpuslocked(state, name, startup, teardown)

For cases where a driver or a subsystem has multiple instances and the same CPU hotplug state callbacks need to be invoked for each instance, the CPU hotplug core provides multi-instance support. The advantage over driver specific instance lists is that the instance related functions are fully serialized against CPU hotplug operations and provide the automatic invocations of the state callbacks on add and removal. To set up such a multi-instance state the following function is available:

- cpuhp_setup_state_multi(state, name, startup, teardown)

The @state argument is either a statically allocated state or one of the constants for dynamically allocated states - CPUHP_BP_PREPARE_DYN, CPUHP_AP_ONLINE_DYN - depending on the state section (PREPARE, ONLINE) for which a dynamic state should be allocated.

The @name argument is used for sysfs output and for instrumentation. The naming convention is "subsys:mode" or "subsys/driver:mode", e.g. "perf:mode" or "perf/x86:mode". The common mode names are:

prepare	For states in the PREPARE section
dead	For states in the PREPARE section which do not provide a startup callback
starting	For states in the STARTING section
dying	For states in the STARTING section which do not provide a startup callback
online	For states in the ONLINE section
offline	For states in the ONLINE section which do not provide a startup callback

As the @name argument is only used for sysfs and instrumentation other mode descriptors can be used as well if they describe the nature of the state better than the common ones.

Examples for @name arguments: "perf/online", "perf/x86:prepare", "RCU/tree:dying", "sched/waitempty"

The @startup argument is a function pointer to the callback which should be invoked during a CPU online operation. If the usage site does not require a startup callback set the pointer to NULL.

The @teardown argument is a function pointer to the callback which should be invoked during a CPU offline operation. If the usage site does not require a teardown callback set the pointer to NULL.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_setup_state_nocalls(), cpuhp_setup_state_nocalls_cpuslocked() and cpuhp_setup_state_multi() only install the callbacks
- cpuhp_setup_state() and cpuhp_setup_state_cpuslocked() install the callbacks and invoke the @startup callback (if not NULL) for all online CPUs which have currently a state greater

than the newly installed state. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

If a callback fails for CPU N then the teardown callback for CPU 0 .. N-1 is invoked to rollback the operation. The state setup fails, the callbacks for the state are not installed and in case of dynamic allocation the allocated state is freed.

The state setup and the callback invocations are serialized against CPU hotplug operations. If the setup function has to be called from a CPU hotplug read locked region, then the _cpuslocked() variants have to be used. These functions cannot be used from within CPU hotplug callbacks.

The function return values:

0	Statically allocated state was successfully set up
>0	Dynamically allocated state was successfully set up. The returned number is the state number which was allocated. If the state callbacks have to be removed later, e.g. module removal, then this number has to be saved by the caller and used as @state argument for the state remove function. For multi-instance states the dynamically allocated state number is also required as @state argument for the instance add/remove operations.
<0	Operation failed

Removal of a CPU hotplug state

To remove a previously set up state, the following functions are provided:

- cpuhp_remove_state(state)
- cpuhp_remove_state_nocalls(state)
- cpuhp_remove_state_nocalls_cpuslocked(state)
- cpuhp_remove_multi_state(state)

The @state argument is either a statically allocated state or the state number which was allocated in the dynamic range by cpuhp_setup_state*(). If the state is in the dynamic range, then the state number is freed and available for dynamic allocation again.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_remove_state_nocalls(), cpuhp_remove_state_nocalls_cpuslocked() and cpuhp_remove_multi_state() only remove the callbacks.
- cpuhp_remove_state() removes the callbacks and invokes the teardown callback (if not NULL) for all online CPUs which have currently a state greater than the removed state. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  In order to complete the removal, the teardown callback should not fail.

The state removal and the callback invocations are serialized against CPU hotplug operations. If the remove function has to be called from a CPU hotplug read locked region, then the _cpuslocked() variants have to be used. These functions cannot be used from within CPU hotplug callbacks.

If a multi-instance state is removed then the caller has to remove all instances first.

Multi-Instance state instance management

Once the multi-instance state is set up, instances can be added to the state:

- cpuhp_state_add_instance(state, node)
- cpuhp_state_add_instance_nocalls(state, node)

The @state argument is either a statically allocated state or the state number which was allocated in the dynamic range by cpuhp_setup_state_multi().

The @node argument is a pointer to an hlist_node which is embedded in the instance's data structure. The pointer is handed to the multi-instance state callbacks and can be used by the callback to retrieve the instance via container_of().

The functions differ in the way how the installed callbacks are treated:

- cpuhp_state_add_instance_nocalls() and only adds the instance to the multi-instance state's node list.
- cpuhp_state_add_instance() adds the instance and invokes the startup callback (if not NULL) associated with @state for all online CPUs which have currently a state greater than @state. The callback is only invoked for the to be added instance. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  If a callback fails for CPU N then the teardown callback for CPU 0 .. N-1 is invoked to rollback the operation, the function fails and the instance is not added to the node list of the multi-instance state.

To remove an instance from the state's node list these functions are available:

- cpuhp_state_remove_instance(state, node)
- cpuhp_state_remove_instance_nocalls(state, node)

The arguments are the same as for the cpuhp_state_add_instance*() variants above.

The functions differ in the way how the installed callbacks are treated:

- cpuhp_state_remove_instance_nocalls() only removes the instance from the state's node list.
- cpuhp_state_remove_instance() removes the instance and invokes the teardown callback (if not NULL) associated with @state for all online CPUs which have currently a state greater than @state. The callback is only invoked for the to be removed instance. Depending on the state section the callback is either invoked on the current CPU (PREPARE section) or on each online CPU (ONLINE section) in the context of the CPU's hotplug thread.

  In order to complete the removal, the teardown callback should not fail.

The node list add/remove operations and the callback invocations are serialized against CPU hotplug operations. These functions cannot be used from within CPU hotplug callbacks and CPU hotplug read locked regions.

Examples

Setup and teardown a statically allocated state in the STARTING section for notifications on online and offline operations:

```
ret = cpuhp_setup_state(CPUHP_SUBSYS_STARTING, "subsys:starting", subsys_cpu_
↪starting, subsys_cpu_dying);
if (ret < 0)
     return ret;
....
cpuhp_remove_state(CPUHP_SUBSYS_STARTING);
```

Setup and teardown a dynamically allocated state in the ONLINE section for notifications on offline operations:

```
state = cpuhp_setup_state(CPUHP_AP_ONLINE_DYN, "subsys:offline", NULL, subsys_
↪cpu_offline);
if (state < 0)
    return state;
....
cpuhp_remove_state(state);
```

Setup and teardown a dynamically allocated state in the ONLINE section for notifications on online operations without invoking the callbacks:

```
state = cpuhp_setup_state_nocalls(CPUHP_AP_ONLINE_DYN, "subsys:online", subsys_
↪cpu_online, NULL);
if (state < 0)
    return state;
....
cpuhp_remove_state_nocalls(state);
```

Setup, use and teardown a dynamically allocated multi-instance state in the ONLINE section for notifications on online and offline operation:

```
state = cpuhp_setup_state_multi(CPUHP_AP_ONLINE_DYN, "subsys:online", subsys_
↪cpu_online, subsys_cpu_offline);
if (state < 0)
    return state;
....
ret = cpuhp_state_add_instance(state, &inst1->node);
if (ret)
     return ret;
....
ret = cpuhp_state_add_instance(state, &inst2->node);
if (ret)
     return ret;
....
cpuhp_remove_instance(state, &inst1->node);
....
cpuhp_remove_instance(state, &inst2->node);
....
```

```
remove_multi_state(state);
```

5.2.7 Testing of hotplug states

One way to verify whether a custom state is working as expected or not is to shutdown a CPU and then put it online again. It is also possible to put the CPU to certain state (for instance *CPUHP_AP_ONLINE*) and then go back to *CPUHP_ONLINE*. This would simulate an error one state after *CPUHP_AP_ONLINE* which would lead to rollback to the online state.

All registered states are enumerated in `/sys/devices/system/cpu/hotplug/states`

```
$ tail /sys/devices/system/cpu/hotplug/states
138: mm/vmscan:online
139: mm/vmstat:online
140: lib/percpu_cnt:online
141: acpi/cpu-drv:online
142: base/cacheinfo:online
143: virtio/net:online
144: x86/mce:online
145: printk:online
168: sched:active
169: online
```

To rollback CPU4 to `lib/percpu_cnt:online` and back online just issue:

```
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
$ echo 140 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
140
```

It is important to note that the teardown callback of state 140 have been invoked. And now get back online:

```
$ echo 169 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
```

With trace events enabled, the individual steps are visible, too:

```
#  TASK-PID   CPU#    TIMESTAMP  FUNCTION
#     | |       |         |         |
    bash-394  [001]  22.976: cpuhp_enter: cpu: 0004 target: 140 step: 169␣
↪(cpuhp_kick_ap_work)
 cpuhp/4-31   [004]  22.977: cpuhp_enter: cpu: 0004 target: 140 step: 168␣
↪(sched_cpu_deactivate)
 cpuhp/4-31   [004]  22.990: cpuhp_exit:  cpu: 0004  state: 168 step: 168 ret:␣
↪0
 cpuhp/4-31   [004]  22.991: cpuhp_enter: cpu: 0004 target: 140 step: 144 (mce_
↪cpu_pre_down)
 cpuhp/4-31   [004]  22.992: cpuhp_exit:  cpu: 0004  state: 144 step: 144 ret:␣
```

```
↪0
cpuhp/4-31   [004]  22.993: cpuhp_multi_enter: cpu: 0004 target: 140 step:␣
↪143 (virtnet_cpu_down_prep)
cpuhp/4-31   [004]  22.994: cpuhp_exit:  cpu: 0004  state: 143 step: 143 ret:␣
↪0
cpuhp/4-31   [004]  22.995: cpuhp_enter: cpu: 0004 target: 140 step: 142␣
↪(cacheinfo_cpu_pre_down)
cpuhp/4-31   [004]  22.996: cpuhp_exit:  cpu: 0004  state: 142 step: 142 ret:␣
↪0
   bash-394  [001]  22.997: cpuhp_exit:  cpu: 0004  state: 140 step: 169 ret:␣
↪0
   bash-394  [005]  95.540: cpuhp_enter: cpu: 0004 target: 169 step: 140␣
↪(cpuhp_kick_ap_work)
cpuhp/4-31   [004]  95.541: cpuhp_enter: cpu: 0004 target: 169 step: 141␣
↪(acpi_soft_cpu_online)
cpuhp/4-31   [004]  95.542: cpuhp_exit:  cpu: 0004  state: 141 step: 141 ret:␣
↪0
cpuhp/4-31   [004]  95.543: cpuhp_enter: cpu: 0004 target: 169 step: 142␣
↪(cacheinfo_cpu_online)
cpuhp/4-31   [004]  95.544: cpuhp_exit:  cpu: 0004  state: 142 step: 142 ret:␣
↪0
cpuhp/4-31   [004]  95.545: cpuhp_multi_enter: cpu: 0004 target: 169 step:␣
↪143 (virtnet_cpu_online)
cpuhp/4-31   [004]  95.546: cpuhp_exit:  cpu: 0004  state: 143 step: 143 ret:␣
↪0
cpuhp/4-31   [004]  95.547: cpuhp_enter: cpu: 0004 target: 169 step: 144 (mce_
↪cpu_online)
cpuhp/4-31   [004]  95.548: cpuhp_exit:  cpu: 0004  state: 144 step: 144 ret:␣
↪0
cpuhp/4-31   [004]  95.549: cpuhp_enter: cpu: 0004 target: 169 step: 145␣
↪(console_cpu_notify)
cpuhp/4-31   [004]  95.550: cpuhp_exit:  cpu: 0004  state: 145 step: 145 ret:␣
↪0
cpuhp/4-31   [004]  95.551: cpuhp_enter: cpu: 0004 target: 169 step: 168␣
↪(sched_cpu_activate)
cpuhp/4-31   [004]  95.552: cpuhp_exit:  cpu: 0004  state: 168 step: 168 ret:␣
↪0
   bash-394  [005]  95.553: cpuhp_exit:  cpu: 0004  state: 169 step: 140 ret:␣
↪0
```

As it an be seen, CPU4 went down until timestamp 22.996 and then back up until 95.552. All invoked callbacks including their return codes are visible in the trace.

5.2.8 Architecture's requirements

The following functions and configurations are required:

CONFIG_HOTPLUG_CPU
This entry needs to be enabled in Kconfig

__cpu_up()
Arch interface to bring up a CPU

__cpu_disable()
Arch interface to shutdown a CPU, no more interrupts can be handled by the kernel after the routine returns. This includes the shutdown of the timer.

__cpu_die()
This actually supposed to ensure death of the CPU. Actually look at some example code in other arch that implement CPU hotplug. The processor is taken down from the `idle()` loop for that specific architecture. `__cpu_die()` typically waits for some per_cpu state to be set, to ensure the processor dead routine is called to be sure positively.

5.2.9 User Space Notification

After CPU successfully onlined or offline udev events are sent. A udev rule like:

```
SUBSYSTEM=="cpu", DRIVERS=="processor", DEVPATH=="/devices/system/cpu/*", RUN+=
↪"the_hotplug_receiver.sh"
```

will receive all events. A script like:

```
#!/bin/sh

if [ "${ACTION}" = "offline" ]
then
    echo "CPU ${DEVPATH##*/} offline"

elif [ "${ACTION}" = "online" ]
then
    echo "CPU ${DEVPATH##*/} online"

fi
```

can process the event further.

When changes to the CPUs in the system occur, the sysfs file /sys/devices/system/cpu/crash_hotplug contains '1' if the kernel updates the kdump capture kernel list of CPUs itself (via elfcorehdr), or '0' if userspace must update the kdump capture kernel list of CPUs.

The availability depends on the CONFIG_HOTPLUG_CPU kernel configuration option.

To skip userspace processing of CPU hot un/plug events for kdump (i.e. the unload-then-reload to obtain a current list of CPUs), this sysfs file can be used in a udev rule as follows:

SUBSYSTEM=="cpu", ATTRS{crash_hotplug}=="1", GOTO="kdump_reload_end"

For a CPU hot un/plug event, if the architecture supports kernel updates of the elfcorehdr (which contains the list of CPUs), then the rule skips the unload-then-reload of the kdump capture kernel.

5.2.10 Kernel Inline Documentations Reference

int **cpuhp_setup_state**(enum cpuhp_state state, const char *name, int (*startup)(unsigned int cpu), int (*teardown)(unsigned int cpu))

Setup hotplug state callbacks with calling the **startup** callback

Parameters

enum cpuhp_state state
The state for which the calls are installed

const char *name
Name of the callback (will be used in debug output)

int (*startup)(unsigned int cpu)
startup callback function or NULL if not required

int (*teardown)(unsigned int cpu)
teardown callback function or NULL if not required

Description

Installs the callback functions and invokes the **startup** callback on the online cpus which have already reached the **state**.

int **cpuhp_setup_state_cpuslocked**(enum cpuhp_state state, const char *name, int (*startup)(unsigned int cpu), int (*teardown)(unsigned int cpu))

Setup hotplug state callbacks with calling **startup** callback from a cpus_read_lock() held region

Parameters

enum cpuhp_state state
The state for which the calls are installed

const char *name
Name of the callback (will be used in debug output)

int (*startup)(unsigned int cpu)
startup callback function or NULL if not required

int (*teardown)(unsigned int cpu)
teardown callback function or NULL if not required

Description

Same as cpuhp_setup_state() except that it must be invoked from within a cpus_read_lock() held region.

int **cpuhp_setup_state_nocalls**(enum cpuhp_state state, const char *name, int (*startup)(unsigned int cpu), int (*teardown)(unsigned int cpu))

Setup hotplug state callbacks without calling the **startup** callback

Parameters

enum cpuhp_state state
The state for which the calls are installed

const char *name
Name of the callback.

int (*startup)(unsigned int cpu)
startup callback function or NULL if not required

int (*teardown)(unsigned int cpu)
teardown callback function or NULL if not required

Description

Same as cpuhp_setup_state() except that the **startup** callback is not invoked during installation. NOP if SMP=n or HOTPLUG_CPU=n.

int **cpuhp_setup_state_nocalls_cpuslocked**(enum cpuhp_state state, const char *name, int (*startup)(unsigned int cpu), int (*teardown)(unsigned int cpu))

Setup hotplug state callbacks without invoking the **startup** callback from a cpus_read_lock() held region callbacks

Parameters

enum cpuhp_state state
The state for which the calls are installed

const char *name
Name of the callback.

int (*startup)(unsigned int cpu)
startup callback function or NULL if not required

int (*teardown)(unsigned int cpu)
teardown callback function or NULL if not required

Description

Same as cpuhp_setup_state_nocalls() except that it must be invoked from within a cpus_read_lock() held region.

int **cpuhp_setup_state_multi**(enum cpuhp_state state, const char *name, int (*startup)(unsigned int cpu, struct hlist_node *node), int (*teardown)(unsigned int cpu, struct hlist_node *node))

Add callbacks for multi state

Parameters

enum cpuhp_state state
The state for which the calls are installed

const char *name
Name of the callback.

int (*startup)(unsigned int cpu, struct hlist_node *node)
startup callback function or NULL if not required

int (*teardown)(unsigned int cpu, struct hlist_node *node)
teardown callback function or NULL if not required

Description

Sets the internal multi_instance flag and prepares a state to work as a multi instance callback. No callbacks are invoked at this point. The callbacks are invoked once an instance for this state are registered via cpuhp_state_add_instance() or cpuhp_state_add_instance_nocalls()

int **cpuhp_state_add_instance**(enum cpuhp_state state, struct hlist_node *node)
Add an instance for a state and invoke startup callback.

Parameters

enum cpuhp_state state
The state for which the instance is installed

struct hlist_node *node
The node for this individual state.

Description

Installs the instance for the **state** and invokes the registered startup callback on the online cpus which have already reached the **state**. The **state** must have been earlier marked as multi-instance by cpuhp_setup_state_multi().

int **cpuhp_state_add_instance_nocalls**(enum cpuhp_state state, struct hlist_node *node)
Add an instance for a state without invoking the startup callback.

Parameters

enum cpuhp_state state
The state for which the instance is installed

struct hlist_node *node
The node for this individual state.

Description

Installs the instance for the **state**. The **state** must have been earlier marked as multi-instance by cpuhp_setup_state_multi. NOP if SMP=n or HOTPLUG_CPU=n.

int **cpuhp_state_add_instance_nocalls_cpuslocked**(enum cpuhp_state state, struct hlist_node *node)
Add an instance for a state without invoking the startup callback from a cpus_read_lock() held region.

Parameters

enum cpuhp_state state
The state for which the instance is installed

struct hlist_node *node
The node for this individual state.

Description

Same as cpuhp_state_add_instance_nocalls() except that it must be invoked from within a cpus_read_lock() held region.

void **cpuhp_remove_state**(enum cpuhp_state state)

Remove hotplug state callbacks and invoke the teardown

Parameters

enum cpuhp_state state
The state for which the calls are removed

Description

Removes the callback functions and invokes the teardown callback on the online cpus which have already reached the **state**.

void **cpuhp_remove_state_nocalls**(enum cpuhp_state state)

Remove hotplug state callbacks without invoking the teardown callback

Parameters

enum cpuhp_state state
The state for which the calls are removed

void **cpuhp_remove_state_nocalls_cpuslocked**(enum cpuhp_state state)

Remove hotplug state callbacks without invoking teardown from a cpus_read_lock() held region.

Parameters

enum cpuhp_state state
The state for which the calls are removed

Description

Same as cpuhp_remove_state nocalls() except that it must be invoked from within a cpus_read_lock() held region.

void **cpuhp_remove_multi_state**(enum cpuhp_state state)

Remove hotplug multi state callback

Parameters

enum cpuhp_state state
The state for which the calls are removed

Description

Removes the callback functions from a multi state. This is the reverse of cpuhp_setup_state_multi(). All instances should have been removed before invoking this function.

int **cpuhp_state_remove_instance**(enum cpuhp_state state, struct hlist_node *node)

Remove hotplug instance from state and invoke the teardown callback

Parameters

enum cpuhp_state state
The state from which the instance is removed

struct hlist_node *node
The node for this individual state.

Description

Removes the instance and invokes the teardown callback on the online cpus which have already reached **state**.

int **cpuhp_state_remove_instance_nocalls**(enum cpuhp_state state, struct hlist_node *node)

Remove hotplug instance from state without invoking the teardown callback

Parameters

enum cpuhp_state state
The state from which the instance is removed

struct hlist_node *node
The node for this individual state.

Description

Removes the instance without invoking the teardown callback.

5.3 Memory hotplug

5.3.1 Memory hotplug event notifier

Hotplugging events are sent to a notification queue.

There are six types of notification defined in `include/linux/memory.h`:

MEM_GOING_ONLINE
Generated before new memory becomes available in order to be able to prepare subsystems to handle memory. The page allocator is still unable to allocate from the new memory.

MEM_CANCEL_ONLINE
Generated if MEM_GOING_ONLINE fails.

MEM_ONLINE
Generated when memory has successfully brought online. The callback may allocate pages from the new memory.

MEM_GOING_OFFLINE
Generated to begin the process of offlining memory. Allocations are no longer possible from the memory but some of the memory to be offlined is still in use. The callback can be used to free memory known to a subsystem from the indicated memory block.

MEM_CANCEL_OFFLINE
Generated if MEM_GOING_OFFLINE fails. Memory is available again from the memory block that we attempted to offline.

MEM_OFFLINE
Generated after offlining memory is complete.

A callback routine can be registered by calling:

```
hotplug_memory_notifier(callback_func, priority)
```

Callback functions with higher values of priority are called before callback functions with lower values.

A callback function must have the following prototype:

```
int callback_func(
  struct notifier_block *self, unsigned long action, void *arg);
```

The first argument of the callback function (self) is a pointer to the block of the notifier chain that points to the callback function itself. The second argument (action) is one of the event types described above. The third argument (arg) passes a pointer of struct memory_notify:

```
struct memory_notify {
        unsigned long start_pfn;
        unsigned long nr_pages;
        int status_change_nid_normal;
        int status_change_nid;
}
```

- start_pfn is start_pfn of online/offline memory.
- nr_pages is # of pages of online/offline memory.
- status_change_nid_normal is set node id when N_NORMAL_MEMORY of nodemask is (will be) set/clear, if this is -1, then nodemask status is not changed.
- status_change_nid is set node id when N_MEMORY of nodemask is (will be) set/clear. It means a new(memoryless) node gets new memory by online and a node loses all memory. If this is -1, then nodemask status is not changed.

  If status_changed_nid* >= 0, callback should create/discard structures for the node if necessary.

The callback routine shall return one of the values NOTIFY_DONE, NOTIFY_OK, NOTIFY_BAD, NOTIFY_STOP defined in `include/linux/notifier.h`

NOTIFY_DONE and NOTIFY_OK have no effect on the further processing.

NOTIFY_BAD is used as response to the MEM_GOING_ONLINE, MEM_GOING_OFFLINE, MEM_ONLINE, or MEM_OFFLINE action to cancel hotplugging. It stops further processing of the notification queue.

NOTIFY_STOP stops further processing of the notification queue.

5.3.2 Locking Internals

When adding/removing memory that uses memory block devices (i.e. ordinary RAM), the device_hotplug_lock should be held to:

- synchronize against online/offline requests (e.g. via sysfs). This way, memory block devices can only be accessed (.online/.state attributes) by user space once memory has been fully added. And when removing memory, we know nobody is in critical sections.
- synchronize against CPU hotplug and similar (e.g. relevant for ACPI and PPC)

Especially, there is a possible lock inversion that is avoided using device_hotplug_lock when adding memory and user space tries to online that memory faster than expected:

- device_online() will first take the device_lock(), followed by mem_hotplug_lock
- add_memory_resource() will first take the mem_hotplug_lock, followed by the device_lock() (while creating the devices, during bus_add_device()).

As the device is visible to user space before taking the device_lock(), this can result in a lock inversion.

onlining/offlining of memory should be done via device_online()/ device_offline() - to make sure it is properly synchronized to actions via sysfs. Holding device_hotplug_lock is advised (to e.g. protect online_type)

When adding/removing/onlining/offlining memory or adding/removing heterogeneous/device memory, we should always hold the mem_hotplug_lock in write mode to serialise memory hotplug (e.g. access to global/zone variables).

In addition, mem_hotplug_lock (in contrast to device_hotplug_lock) in read mode allows for a quite efficient get_online_mems/put_online_mems implementation, so code accessing memory can protect from that memory vanishing.

5.4 Linux generic IRQ handling

5.4.1 Introduction

The generic interrupt handling layer is designed to provide a complete abstraction of interrupt handling for device drivers. It is able to handle all the different types of interrupt controller hardware. Device drivers use generic API functions to request, enable, disable and free interrupts. The drivers do not have to know anything about interrupt hardware details, so they can be used on different platforms without code changes.

This documentation is provided to developers who want to implement an interrupt subsystem based for their architecture, with the help of the generic IRQ handling layer.

5.4.2 Rationale

The original implementation of interrupt handling in Linux uses the __do_IRQ() super-handler, which is able to deal with every type of interrupt logic.

Originally, Russell King identified different types of handlers to build a quite universal set for the ARM interrupt handler implementation in Linux 2.5/2.6. He distinguished between:

- Level type
- Edge type
- Simple type

During the implementation we identified another type:

- Fast EOI type

In the SMP world of the __do_IRQ() super-handler another type was identified:

- Per CPU type

This split implementation of high-level IRQ handlers allows us to optimize the flow of the interrupt handling for each specific interrupt type. This reduces complexity in that particular code path and allows the optimized handling of a given type.

The original general IRQ implementation used hw_interrupt_type structures and their `->ack`, `->end` [etc.] callbacks to differentiate the flow control in the super-handler. This leads to a mix of flow logic and low-level hardware logic, and it also leads to unnecessary code duplication: for example in i386, there is an `ioapic_level_irq` and an `ioapic_edge_irq` IRQ-type which share many of the low-level details but have different flow handling.

A more natural abstraction is the clean separation of the 'irq flow' and the 'chip details'.

Analysing a couple of architecture's IRQ subsystem implementations reveals that most of them can use a generic set of 'irq flow' methods and only need to add the chip-level specific code. The separation is also valuable for (sub)architectures which need specific quirks in the IRQ flow itself but not in the chip details - and thus provides a more transparent IRQ subsystem design.

Each interrupt descriptor is assigned its own high-level flow handler, which is normally one of the generic implementations. (This high-level flow handler implementation also makes it simple to provide demultiplexing handlers which can be found in embedded platforms on various architectures.)

The separation makes the generic interrupt handling layer more flexible and extensible. For example, an (sub)architecture can use a generic IRQ-flow implementation for 'level type' interrupts and add a (sub)architecture specific 'edge type' implementation.

To make the transition to the new model easier and prevent the breakage of existing implementations, the __do_IRQ() super-handler is still available. This leads to a kind of duality for the time being. Over time the new model should be used in more and more architectures, as it enables smaller and cleaner IRQ subsystems. It's deprecated for three years now and about to be removed.

5.4.3 Known Bugs And Assumptions

None (knock on wood).

5.4.4 Abstraction layers

There are three main levels of abstraction in the interrupt code:

1. High-level driver API
2. High-level IRQ flow handlers
3. Chip-level hardware encapsulation

Interrupt control flow

Each interrupt is described by an interrupt descriptor structure irq_desc. The interrupt is referenced by an 'unsigned int' numeric value which selects the corresponding interrupt description structure in the descriptor structures array. The descriptor structure contains status information and pointers to the interrupt flow method and the interrupt chip structure which are assigned to this interrupt.

Whenever an interrupt triggers, the low-level architecture code calls into the generic interrupt code by calling desc->handle_irq(). This high-level IRQ handling function only uses desc->irq_data.chip primitives referenced by the assigned chip descriptor structure.

High-level Driver API

The high-level Driver API consists of following functions:

- request_irq()
- request_threaded_irq()
- free_irq()
- disable_irq()
- enable_irq()
- disable_irq_nosync() (SMP only)
- synchronize_irq() (SMP only)
- irq_set_irq_type()
- irq_set_irq_wake()
- irq_set_handler_data()
- irq_set_chip()
- irq_set_chip_data()

See the autogenerated function documentation for details.

High-level IRQ flow handlers

The generic layer provides a set of pre-defined irq-flow methods:

- handle_level_irq()
- handle_edge_irq()
- handle_fasteoi_irq()
- handle_simple_irq()
- handle_percpu_irq()
- handle_edge_eoi_irq()
- handle_bad_irq()

The interrupt flow handlers (either pre-defined or architecture specific) are assigned to specific interrupts by the architecture either during bootup or during device initialization.

Default flow implementations

Helper functions

The helper functions call the chip primitives and are used by the default flow implementations. The following helper functions are implemented (simplified excerpt):

```
default_enable(struct irq_data *data)
{
    desc->irq_data.chip->irq_unmask(data);
}

default_disable(struct irq_data *data)
{
    if (!delay_disable(data))
        desc->irq_data.chip->irq_mask(data);
}

default_ack(struct irq_data *data)
{
    chip->irq_ack(data);
}

default_mask_ack(struct irq_data *data)
{
    if (chip->irq_mask_ack) {
        chip->irq_mask_ack(data);
    } else {
        chip->irq_mask(data);
        chip->irq_ack(data);
    }
}

noop(struct irq_data *data))
{
}
```

Default flow handler implementations

Default Level IRQ flow handler

handle_level_irq provides a generic implementation for level-triggered interrupts.

The following control flow is implemented (simplified excerpt):

```
desc->irq_data.chip->irq_mask_ack();
handle_irq_event(desc->action);
desc->irq_data.chip->irq_unmask();
```

Default Fast EOI IRQ flow handler

handle_fasteoi_irq provides a generic implementation for interrupts, which only need an EOI at the end of the handler.

The following control flow is implemented (simplified excerpt):

```
handle_irq_event(desc->action);
desc->irq_data.chip->irq_eoi();
```

Default Edge IRQ flow handler

handle_edge_irq provides a generic implementation for edge-triggered interrupts.

The following control flow is implemented (simplified excerpt):

```
if (desc->status & running) {
    desc->irq_data.chip->irq_mask_ack();
    desc->status |= pending | masked;
    return;
}
desc->irq_data.chip->irq_ack();
desc->status |= running;
do {
    if (desc->status & masked)
        desc->irq_data.chip->irq_unmask();
    desc->status &= ~pending;
    handle_irq_event(desc->action);
} while (desc->status & pending);
desc->status &= ~running;
```

Default simple IRQ flow handler

handle_simple_irq provides a generic implementation for simple interrupts.

Note: The simple flow handler does not call any handler/chip primitives.

The following control flow is implemented (simplified excerpt):

```
handle_irq_event(desc->action);
```

Default per CPU flow handler

handle_percpu_irq provides a generic implementation for per CPU interrupts.

Per CPU interrupts are only available on SMP and the handler provides a simplified version without locking.

The following control flow is implemented (simplified excerpt):

```
if (desc->irq_data.chip->irq_ack)
    desc->irq_data.chip->irq_ack();
handle_irq_event(desc->action);
if (desc->irq_data.chip->irq_eoi)
    desc->irq_data.chip->irq_eoi();
```

EOI Edge IRQ flow handler

handle_edge_eoi_irq provides an abnomination of the edge handler which is solely used to tame a badly wreckaged irq controller on powerpc/cell.

Bad IRQ flow handler

handle_bad_irq is used for spurious interrupts which have no real handler assigned..

Quirks and optimizations

The generic functions are intended for 'clean' architectures and chips, which have no platform-specific IRQ handling quirks. If an architecture needs to implement quirks on the 'flow' level then it can do so by overriding the high-level irq-flow handler.

Delayed interrupt disable

This per interrupt selectable feature, which was introduced by Russell King in the ARM interrupt implementation, does not mask an interrupt at the hardware level when disable_irq() is called. The interrupt is kept enabled and is masked in the flow handler when an interrupt event happens. This prevents losing edge interrupts on hardware which does not store an edge interrupt event while the interrupt is disabled at the hardware level. When an interrupt arrives while the IRQ_DISABLED flag is set, then the interrupt is masked at the hardware level and the IRQ_PENDING bit is set. When the interrupt is re-enabled by enable_irq() the pending bit is checked and if it is set, the interrupt is resent either via hardware or by a software resend mechanism. (It's necessary to enable CONFIG_HARDIRQS_SW_RESEND when you want to use the delayed interrupt disable feature and your hardware is not capable of retriggering an interrupt.) The delayed interrupt disable is not configurable.

Chip-level hardware encapsulation

The chip-level hardware descriptor structure *irq_chip* contains all the direct chip relevant functions, which can be utilized by the irq flow implementations.

- `irq_ack`
- `irq_mask_ack` - Optional, recommended for performance
- `irq_mask`
- `irq_unmask`
- `irq_eoi` - Optional, required for EOI flow handlers
- `irq_retrigger` - Optional
- `irq_set_type` - Optional
- `irq_set_wake` - Optional

These primitives are strictly intended to mean what they say: ack means ACK, masking means masking of an IRQ line, etc. It is up to the flow handler(s) to use these basic units of low-level functionality.

5.4.5 __do_IRQ entry point

The original implementation __do_IRQ() was an alternative entry point for all types of interrupts. It no longer exists.

This handler turned out to be not suitable for all interrupt hardware and was therefore reimplemented with split functionality for edge/level/simple/percpu interrupts. This is not only a functional optimization. It also shortens code paths for interrupts.

5.4.6 Locking on SMP

The locking of chip registers is up to the architecture that defines the chip primitives. The per-irq structure is protected via desc->lock, by the generic layer.

5.4.7 Generic interrupt chip

To avoid copies of identical implementations of IRQ chips the core provides a configurable generic interrupt chip implementation. Developers should check carefully whether the generic chip fits their needs before implementing the same functionality slightly differently themselves.

void **irq_gc_noop**(struct *irq_data* *d)
NOOP function

Parameters

struct irq_data *d
irq_data

void **irq_gc_mask_disable_reg**(struct *irq_data* *d)
Mask chip via disable register

Parameters

struct irq_data *d
irq_data

Description

Chip has separate enable/disable registers instead of a single mask register.

void **irq_gc_mask_set_bit**(struct *irq_data* *d)
Mask chip via setting bit in mask register

Parameters

struct irq_data *d
irq_data

Description

Chip has a single mask register. Values of this register are cached and protected by gc->lock

void **irq_gc_mask_clr_bit**(struct *irq_data* *d)
Mask chip via clearing bit in mask register

Parameters

struct irq_data *d
irq_data

Description

Chip has a single mask register. Values of this register are cached and protected by gc->lock

void **irq_gc_unmask_enable_reg**(struct *irq_data* *d)
Unmask chip via enable register

Parameters

struct irq_data *d
irq_data

Description

Chip has separate enable/disable registers instead of a single mask register.

void **irq_gc_ack_set_bit**(struct *irq_data* *d)
Ack pending interrupt via setting bit

Parameters

struct irq_data *d
irq_data

int **irq_gc_set_wake**(struct *irq_data* *d, unsigned int on)
Set/clr wake bit for an interrupt

Parameters

struct irq_data *d
irq_data

unsigned int on
Indicates whether the wake bit should be set or cleared

Description

For chips where the wake from suspend functionality is not configured in a separate register and the wakeup active state is just stored in a bitmask.

struct *irq_chip_generic* ***irq_alloc_generic_chip**(const char *name, int num_ct, unsigned int irq_base, void __iomem *reg_base, irq_flow_handler_t handler)
Allocate a generic chip and initialize it

Parameters

const char *name
Name of the irq chip

int num_ct
Number of irq_chip_type instances associated with this

unsigned int irq_base
Interrupt base nr for this chip

void __iomem *reg_base
Register base address (virtual)

irq_flow_handler_t handler
Default flow handler associated with this chip

Description

Returns an initialized irq_chip_generic structure. The chip defaults to the primary (index 0) irq_chip_type and **handler**

int **__irq_alloc_domain_generic_chips**(struct irq_domain *d, int irqs_per_chip, int num_ct, const char *name, irq_flow_handler_t handler, unsigned int clr, unsigned int set, enum *irq_gc_flags* gcflags)

Allocate generic chips for an irq domain

Parameters

struct irq_domain *d
irq domain for which to allocate chips

int irqs_per_chip
Number of interrupts each chip handles (max 32)

int num_ct
Number of irq_chip_type instances associated with this

const char *name
Name of the irq chip

irq_flow_handler_t handler
Default flow handler associated with these chips

unsigned int clr
IRQ_* bits to clear in the mapping function

unsigned int set
IRQ_* bits to set in the mapping function

enum irq_gc_flags gcflags
Generic chip specific setup flags

struct *irq_chip_generic* ***irq_get_domain_generic_chip**(struct irq_domain *d, unsigned int hw_irq)

Get a pointer to the generic chip of a hw_irq

Parameters

struct irq_domain *d
irq domain pointer

unsigned int hw_irq
Hardware interrupt number

void **irq_setup_generic_chip**(struct *irq_chip_generic* *gc, u32 msk, enum *irq_gc_flags* flags, unsigned int clr, unsigned int set)

Setup a range of interrupts with a generic chip

Parameters

struct irq_chip_generic *gc
Generic irq chip holding all data

u32 msk
Bitmask holding the irqs to initialize relative to gc->irq_base

enum irq_gc_flags flags
Flags for initialization

unsigned int clr
IRQ_* bits to clear

unsigned int set
IRQ_* bits to set

Description

Set up max. 32 interrupts starting from gc->irq_base. Note, this initializes all interrupts to the primary irq_chip_type and its associated handler.

int **irq_setup_alt_chip**(struct *irq_data* *d, unsigned int type)

Switch to alternative chip

Parameters

struct irq_data *d
irq_data for this interrupt

unsigned int type
Flow type to be initialized

Description

Only to be called from chip->irq_set_type() callbacks.

void **irq_remove_generic_chip**(struct *irq_chip_generic* *gc, u32 msk, unsigned int clr, unsigned int set)

Remove a chip

Parameters

struct irq_chip_generic *gc
Generic irq chip holding all data

u32 msk
Bitmask holding the irqs to initialize relative to gc->irq_base

unsigned int clr
IRQ_* bits to clear

unsigned int set
IRQ_* bits to set

Description

Remove up to 32 interrupts starting from gc->irq_base.

5.4.8 Structures

This chapter contains the autogenerated documentation of the structures which are used in the generic IRQ layer.

struct **irq_common_data**

per irq data shared by all irqchips

Definition:

```
struct irq_common_data {
    unsigned int            __private state_use_accessors;
#ifdef CONFIG_NUMA;
    unsigned int            node;
#endif;
    void *handler_data;
```

```
    struct msi_desc         *msi_desc;
#ifdef CONFIG_SMP;
    cpumask_var_t affinity;
#endif;
#ifdef CONFIG_GENERIC_IRQ_EFFECTIVE_AFF_MASK;
    cpumask_var_t effective_affinity;
#endif;
#ifdef CONFIG_GENERIC_IRQ_IPI;
    unsigned int            ipi_offset;
#endif;
};
```

Members

state_use_accessors
status information for irq chip functions. Use accessor functions to deal with it

node
node index useful for balancing

handler_data
per-IRQ data for the irq_chip methods

msi_desc
MSI descriptor

affinity
IRQ affinity on SMP. If this is an IPI related irq, then this is the mask of the CPUs to which an IPI can be sent.

effective_affinity
The effective IRQ affinity on SMP as some irq chips do not allow multi CPU destinations. A subset of **affinity**.

ipi_offset
Offset of first IPI target cpu in **affinity**. Optional.

struct **irq_data**

per irq chip data passed down to chip functions

Definition:

```
struct irq_data {
    u32 mask;
    unsigned int            irq;
    unsigned long           hwirq;
    struct irq_common_data  *common;
    struct irq_chip         *chip;
    struct irq_domain       *domain;
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY;
    struct irq_data         *parent_data;
#endif;
    void *chip_data;
};
```

Members

mask
precomputed bitmask for accessing the chip registers

irq
interrupt number

hwirq
hardware interrupt number, local to the interrupt domain

common
point to data shared by all irqchips

chip
low level interrupt hardware access

domain
Interrupt translation domain; responsible for mapping between hwirq number and linux irq number.

parent_data
pointer to parent *struct irq_data* to support hierarchy irq_domain

chip_data
platform-specific per-chip private data for the chip methods, to allow shared chip implementations

struct **irq_chip**

hardware interrupt chip descriptor

Definition:

```
struct irq_chip {
    const char      *name;
    unsigned int    (*irq_startup)(struct irq_data *data);
    void (*irq_shutdown)(struct irq_data *data);
    void (*irq_enable)(struct irq_data *data);
    void (*irq_disable)(struct irq_data *data);
    void (*irq_ack)(struct irq_data *data);
    void (*irq_mask)(struct irq_data *data);
    void (*irq_mask_ack)(struct irq_data *data);
    void (*irq_unmask)(struct irq_data *data);
    void (*irq_eoi)(struct irq_data *data);
    int (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest,␣
↪bool force);
    int (*irq_retrigger)(struct irq_data *data);
    int (*irq_set_type)(struct irq_data *data, unsigned int flow_type);
    int (*irq_set_wake)(struct irq_data *data, unsigned int on);
    void (*irq_bus_lock)(struct irq_data *data);
    void (*irq_bus_sync_unlock)(struct irq_data *data);
#ifdef CONFIG_DEPRECATED_IRQ_CPU_ONOFFLINE;
    void (*irq_cpu_online)(struct irq_data *data);
    void (*irq_cpu_offline)(struct irq_data *data);
#endif;
    void (*irq_suspend)(struct irq_data *data);
```

```
    void (*irq_resume)(struct irq_data *data);
    void (*irq_pm_shutdown)(struct irq_data *data);
    void (*irq_calc_mask)(struct irq_data *data);
    void (*irq_print_chip)(struct irq_data *data, struct seq_file *p);
    int (*irq_request_resources)(struct irq_data *data);
    void (*irq_release_resources)(struct irq_data *data);
    void (*irq_compose_msi_msg)(struct irq_data *data, struct msi_msg *msg);
    void (*irq_write_msi_msg)(struct irq_data *data, struct msi_msg *msg);
    int (*irq_get_irqchip_state)(struct irq_data *data, enum irqchip_irq_state␣
↪which, bool *state);
    int (*irq_set_irqchip_state)(struct irq_data *data, enum irqchip_irq_state␣
↪which, bool state);
    int (*irq_set_vcpu_affinity)(struct irq_data *data, void *vcpu_info);
    void (*ipi_send_single)(struct irq_data *data, unsigned int cpu);
    void (*ipi_send_mask)(struct irq_data *data, const struct cpumask *dest);
    int (*irq_nmi_setup)(struct irq_data *data);
    void (*irq_nmi_teardown)(struct irq_data *data);
    unsigned long   flags;
};
```

Members

name
name for /proc/interrupts

irq_startup
start up the interrupt (defaults to ->enable if NULL)

irq_shutdown
shut down the interrupt (defaults to ->disable if NULL)

irq_enable
enable the interrupt (defaults to chip->unmask if NULL)

irq_disable
disable the interrupt

irq_ack
start of a new interrupt

irq_mask
mask an interrupt source

irq_mask_ack
ack and mask an interrupt source

irq_unmask
unmask an interrupt source

irq_eoi
end of interrupt

irq_set_affinity
Set the CPU affinity on SMP machines. If the force argument is true, it tells the driver to unconditionally apply the affinity setting. Sanity checks against the supplied affinity mask

are not required. This is used for CPU hotplug where the target CPU is not yet set in the cpu_online_mask.

irq_retrigger
resend an IRQ to the CPU

irq_set_type
set the flow type (IRQ_TYPE_LEVEL/etc.) of an IRQ

irq_set_wake
enable/disable power-management wake-on of an IRQ

irq_bus_lock
function to lock access to slow bus (i2c) chips

irq_bus_sync_unlock
function to sync and unlock slow bus (i2c) chips

irq_cpu_online
configure an interrupt source for a secondary CPU

irq_cpu_offline
un-configure an interrupt source for a secondary CPU

irq_suspend
function called from core code on suspend once per chip, when one or more interrupts are installed

irq_resume
function called from core code on resume once per chip, when one ore more interrupts are installed

irq_pm_shutdown
function called from core code on shutdown once per chip

irq_calc_mask
Optional function to set irq_data.mask for special cases

irq_print_chip
optional to print special chip info in show_interrupts

irq_request_resources
optional to request resources before calling any other callback related to this irq

irq_release_resources
optional to release resources acquired with irq_request_resources

irq_compose_msi_msg
optional to compose message content for MSI

irq_write_msi_msg
optional to write message content for MSI

irq_get_irqchip_state
return the internal state of an interrupt

irq_set_irqchip_state
set the internal state of a interrupt

irq_set_vcpu_affinity
optional to target a vCPU in a virtual machine

ipi_send_single
send a single IPI to destination cpus

ipi_send_mask
send an IPI to destination cpus in cpumask

irq_nmi_setup
function called from core code before enabling an NMI

irq_nmi_teardown
function called from core code after disabling an NMI

flags
chip specific flags

struct **irq_chip_regs**

register offsets for struct irq_gci

Definition:

```
struct irq_chip_regs {
    unsigned long           enable;
    unsigned long           disable;
    unsigned long           mask;
    unsigned long           ack;
    unsigned long           eoi;
    unsigned long           type;
    unsigned long           polarity;
};
```

Members

enable
Enable register offset to reg_base

disable
Disable register offset to reg_base

mask
Mask register offset to reg_base

ack
Ack register offset to reg_base

eoi
Eoi register offset to reg_base

type
Type configuration register offset to reg_base

polarity
Polarity configuration register offset to reg_base

struct **irq_chip_type**

Generic interrupt chip instance for a flow type

Definition:

```
struct irq_chip_type {
    struct irq_chip         chip;
    struct irq_chip_regs    regs;
    irq_flow_handler_t handler;
    u32 type;
    u32 mask_cache_priv;
    u32 *mask_cache;
};
```

Members

chip
The real interrupt chip which provides the callbacks

regs
Register offsets for this chip

handler
Flow handler associated with this chip

type
Chip can handle these flow types

mask_cache_priv
Cached mask register private to the chip type

mask_cache
Pointer to cached mask register

Description

A irq_generic_chip can have several instances of irq_chip_type when it requires different functions and register offsets for different flow types.

struct **irq_chip_generic**

Generic irq chip data structure

Definition:

```
struct irq_chip_generic {
    raw_spinlock_t lock;
    void __iomem            *reg_base;
    u32 (*reg_readl)(void __iomem *addr);
    void (*reg_writel)(u32 val, void __iomem *addr);
    void (*suspend)(struct irq_chip_generic *gc);
    void (*resume)(struct irq_chip_generic *gc);
    unsigned int            irq_base;
    unsigned int            irq_cnt;
    u32 mask_cache;
    u32 type_cache;
    u32 polarity_cache;
    u32 wake_enabled;
    u32 wake_active;
    unsigned int            num_ct;
    void *private;
    unsigned long           installed;
```

```
    unsigned long           unused;
    struct irq_domain       *domain;
    struct list_head        list;
    struct irq_chip_type    chip_types[];
};
```

Members

lock
Lock to protect register and cache data access

reg_base
Register base address (virtual)

reg_readl
Alternate I/O accessor (defaults to readl if NULL)

reg_writel
Alternate I/O accessor (defaults to writel if NULL)

suspend
Function called from core code on suspend once per chip; can be useful instead of irq_chip::suspend to handle chip details even when no interrupts are in use

resume
Function called from core code on resume once per chip; can be useful instead of irq_chip::suspend to handle chip details even when no interrupts are in use

irq_base
Interrupt base nr for this chip

irq_cnt
Number of interrupts handled by this chip

mask_cache
Cached mask register shared between all chip types

type_cache
Cached type register

polarity_cache
Cached polarity register

wake_enabled
Interrupt can wakeup from suspend

wake_active
Interrupt is marked as an wakeup from suspend source

num_ct
Number of available irq_chip_type instances (usually 1)

private
Private data for non generic chip callbacks

installed
bitfield to denote installed interrupts

unused
bitfield to denote unused interrupts

domain
irq domain pointer

list
List head for keeping track of instances

chip_types
Array of interrupt irq_chip_types

Description

Note, that irq_chip_generic can have multiple irq_chip_type implementations which can be associated to a particular irq line of an irq_chip_generic instance. That allows to share and protect state in an irq_chip_generic instance when we need to implement different flow mechanisms (level/edge) for it.

enum **irq_gc_flags**

Initialization flags for generic irq chips

Constants

IRQ_GC_INIT_MASK_CACHE
Initialize the mask_cache by reading mask reg

IRQ_GC_INIT_NESTED_LOCK
Set the lock class of the irqs to nested for irq chips which need to call irq_set_wake() on the parent irq. Usually GPIO implementations

IRQ_GC_MASK_CACHE_PER_TYPE
Mask cache is chip type private

IRQ_GC_NO_MASK
Do not calculate irq_data->mask

IRQ_GC_BE_IO
Use big-endian register accesses (default: LE)

struct **irqaction**

per interrupt action descriptor

Definition:

```
struct irqaction {
    irq_handler_t handler;
    void *dev_id;
    void __percpu           *percpu_dev_id;
    struct irqaction        *next;
    irq_handler_t thread_fn;
    struct task_struct      *thread;
    struct irqaction        *secondary;
    unsigned int            irq;
    unsigned int            flags;
    unsigned long           thread_flags;
    unsigned long           thread_mask;
    const char              *name;
```

```
    struct proc_dir_entry   *dir;
};
```

Members

handler
interrupt handler function

dev_id
cookie to identify the device

percpu_dev_id
cookie to identify the device

next
pointer to the next irqaction for shared interrupts

thread_fn
interrupt handler function for threaded interrupts

thread
thread pointer for threaded interrupts

secondary
pointer to secondary irqaction (force threading)

irq
interrupt number

flags
flags (see IRQF_* above)

thread_flags
flags related to **thread**

thread_mask
bitmask for keeping track of **thread** activity

name
name of the device

dir
pointer to the proc/irq/NN/name entry

int **request_irq**(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)

Add a handler for an interrupt line

Parameters

unsigned int irq
The interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs. Primary handler for threaded interrupts If NULL, the default primary handler is installed

unsigned long flags
Handling flags

const char *name
Name of the device generating this interrupt

void *dev
A cookie passed to the handler function

Description

This call allocates an interrupt and establishes a handler; see the documentation for request_threaded_irq() for details.

struct **irq_affinity_notify**
context for notification of IRQ affinity changes

Definition:

```
struct irq_affinity_notify {
    unsigned int irq;
    struct kref kref;
    struct work_struct work;
    void (*notify)(struct irq_affinity_notify *, const cpumask_t *mask);
    void (*release)(struct kref *ref);
};
```

Members

irq
Interrupt to which notification applies

kref
Reference count, for internal use

work
Work item, for internal use

notify
Function to be called on change. This will be called in process context.

release
Function to be called on release. This will be called in process context. Once registered, the structure must only be freed when this function is called or later.

struct **irq_affinity**
Description for automatic irq affinity assignements

Definition:

```
struct irq_affinity {
    unsigned int    pre_vectors;
    unsigned int    post_vectors;
    unsigned int    nr_sets;
    unsigned int    set_size[IRQ_AFFINITY_MAX_SETS];
    void (*calc_sets)(struct irq_affinity *, unsigned int nvecs);
    void *priv;
};
```

Members

pre_vectors
Don't apply affinity to **pre_vectors** at beginning of the MSI(-X) vector space

post_vectors
Don't apply affinity to **post_vectors** at end of the MSI(-X) vector space

nr_sets
The number of interrupt sets for which affinity spreading is required

set_size
Array holding the size of each interrupt set

calc_sets
Callback for calculating the number and size of interrupt sets

priv
Private data for usage by **calc_sets**, usually a pointer to driver/device specific data.

struct **irq_affinity_desc**
Interrupt affinity descriptor

Definition:

```
struct irq_affinity_desc {
    struct cpumask  mask;
    unsigned int    is_managed : 1;
};
```

Members

mask
cpumask to hold the affinity assignment

is_managed
1 if the interrupt is managed internally

int **irq_update_affinity_hint**(unsigned int irq, const struct cpumask *m)
Update the affinity hint

Parameters

unsigned int irq
Interrupt to update

const struct cpumask *m
cpumask pointer (NULL to clear the hint)

Description

Updates the affinity hint, but does not change the affinity of the interrupt.

int **irq_set_affinity_and_hint**(unsigned int irq, const struct cpumask *m)
Update the affinity hint and apply the provided cpumask to the interrupt

Parameters

unsigned int irq
Interrupt to update

const struct cpumask *m
cpumask pointer (NULL to clear the hint)

Description

Updates the affinity hint and if **m** is not NULL it applies it as the affinity of that interrupt.

5.4.9 Public Functions Provided

This chapter contains the autogenerated documentation of the kernel API functions which are exported.

bool **synchronize_hardirq**(unsigned int irq)
wait for pending hard IRQ handlers (on other CPUs)

Parameters

unsigned int irq
interrupt number to wait for

This function waits for any pending hard IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock. It does not take associated threaded handlers into account.

Do not use this for shutdown scenarios where you must be sure that all parts (hardirq and threaded handler) have completed.

Return

false if a threaded handler is active.

This function may be called - with care - from IRQ context.

It does not check whether there is an interrupt in flight at the hardware level, but not serviced yet, as this might deadlock when called with interrupts disabled and the target CPU of the interrupt is the current CPU.

void **synchronize_irq**(unsigned int irq)
wait for pending IRQ handlers (on other CPUs)

Parameters

unsigned int irq
interrupt number to wait for

This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

Can only be called from preemptible code as it might sleep when an interrupt thread is associated to **irq**.

It optionally makes sure (when the irq chip supports that method) that the interrupt is not pending in any CPU and waiting for service.

int **irq_can_set_affinity**(unsigned int irq)
Check if the affinity of a given irq can be set

Parameters

unsigned int irq
Interrupt to check

bool **irq_can_set_affinity_usr**(unsigned int irq)
Check if affinity of a irq can be set from user space

Parameters

unsigned int irq
Interrupt to check

Description

Like *irq_can_set_affinity()* above, but additionally checks for the AFFINITY_MANAGED flag.

void **irq_set_thread_affinity**(struct irq_desc *desc)
Notify irq threads to adjust affinity

Parameters

struct irq_desc *desc
irq descriptor which has affinity changed

We just set IRQTF_AFFINITY and delegate the affinity setting to the interrupt thread itself. We can not call set_cpus_allowed_ptr() here as we hold desc->lock and this code can be called from hard interrupt context.

int **irq_update_affinity_desc**(unsigned int irq, struct *irq_affinity_desc* *affinity)
Update affinity management for an interrupt

Parameters

unsigned int irq
The interrupt number to update

struct irq_affinity_desc *affinity
Pointer to the affinity descriptor

Description

This interface can be used to configure the affinity management of interrupts which have been allocated already.

There are certain limitations on when it may be used - attempts to use it for when the kernel is configured for generic IRQ reservation mode (in config GENERIC_IRQ_RESERVATION_MODE) will fail, as it may conflict with managed/non-managed interrupt accounting. In addition, attempts to use it on an interrupt which is already started or which has already been configured as managed will also fail, as these mean invalid init state or double init.

int **irq_set_affinity**(unsigned int irq, const struct *cpumask* *cpumask)
Set the irq affinity of a given irq

Parameters

unsigned int irq
Interrupt to set affinity

const struct cpumask *cpumask
cpumask

Description

Fails if cpumask does not contain an online CPU

int **irq_force_affinity**(unsigned int irq, const struct *cpumask* *cpumask)

Force the irq affinity of a given irq

Parameters

unsigned int irq
Interrupt to set affinity

const struct cpumask *cpumask
cpumask

Description

Same as irq_set_affinity, but without checking the mask against online cpus.

Solely for low level cpu hotplug code, where we need to make per cpu interrupts affine before the cpu becomes online.

int **irq_set_affinity_notifier**(unsigned int irq, struct *irq_affinity_notify* *notify)

control notification of IRQ affinity changes

Parameters

unsigned int irq
Interrupt for which to enable/disable notification

struct irq_affinity_notify *notify
Context for notification, or NULL to disable notification. Function pointers must be initialised; the other fields will be initialised by this function.

Must be called in process context. Notification may only be enabled after the IRQ is allocated and must be disabled before the IRQ is freed using free_irq().

int **irq_set_vcpu_affinity**(unsigned int irq, void *vcpu_info)

Set vcpu affinity for the interrupt

Parameters

unsigned int irq
interrupt number to set affinity

void *vcpu_info
vCPU specific data or pointer to a percpu array of vCPU specific data for percpu_devid interrupts

This function uses the vCPU specific data to set the vCPU affinity for an irq. The vCPU specific data is passed from outside, such as KVM. One example code path is as below: KVM -> IOMMU -> *irq_set_vcpu_affinity()*.

void **disable_irq_nosync**(unsigned int irq)

disable an irq without waiting

Parameters

unsigned int irq
Interrupt to disable

Disable the selected interrupt line. Disables and Enables are nested. Unlike disable_irq(), this function does not ensure existing instances of the IRQ handler have completed before returning.

This function may be called from IRQ context.

void **disable_irq**(unsigned int irq)

disable an irq and wait for completion

Parameters

unsigned int irq
Interrupt to disable

Disable the selected interrupt line. Enables and Disables are nested. This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

Can only be called from preemptible code as it might sleep when an interrupt thread is associated to **irq**.

bool **disable_hardirq**(unsigned int irq)

disables an irq and waits for hardirq completion

Parameters

unsigned int irq
Interrupt to disable

Disable the selected interrupt line. Enables and Disables are nested. This function waits for any pending hard IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the hard IRQ handler may need you will deadlock.

When used to optimistically disable an interrupt from atomic context the return value must be checked.

Return

false if a threaded handler is active.

This function may be called - with care - from IRQ context.

void **disable_nmi_nosync**(unsigned int irq)

disable an nmi without waiting

Parameters

unsigned int irq
Interrupt to disable

Disable the selected interrupt line. Disables and enables are nested. The interrupt to disable must have been requested through request_nmi. Unlike disable_nmi(), this function does not ensure existing instances of the IRQ handler have completed before returning.

void **enable_irq**(unsigned int irq)

enable handling of an irq

Parameters

unsigned int irq
Interrupt to enable

Undoes the effect of one call to disable_irq(). If this matches the last disable, processing of interrupts on this IRQ line is re-enabled.

This function may be called from IRQ context only when desc->irq_data.chip->bus_lock and desc->chip->bus_sync_unlock are NULL !

void **enable_nmi**(unsigned int irq)
enable handling of an nmi

Parameters

unsigned int irq
Interrupt to enable

The interrupt to enable must have been requested through request_nmi. Undoes the effect of one call to disable_nmi(). If this matches the last disable, processing of interrupts on this IRQ line is re-enabled.

int **irq_set_irq_wake**(unsigned int irq, unsigned int on)
control irq power management wakeup

Parameters

unsigned int irq
interrupt to control

unsigned int on
enable/disable power management wakeup

Enable/disable power management wakeup mode, which is disabled by default. Enables and disables must match, just as they match for non-wakeup mode support.

Wakeup mode lets this IRQ wake the system from sleep states like "suspend to RAM".

Note

irq enable/disable state is completely orthogonal
to the enable/disable state of irq wake. An irq can be disabled with disable_irq() and still wake the system as long as the irq has wake enabled. If this does not hold, then the underlying irq chip and the related driver need to be investigated.

void **irq_wake_thread**(unsigned int irq, void *dev_id)
wake the irq thread for the action identified by dev_id

Parameters

unsigned int irq
Interrupt line

void *dev_id
Device identity for which the thread should be woken

const void ***free_irq**(unsigned int irq, void *dev_id)
free an interrupt allocated with request_irq

Parameters

unsigned int irq
Interrupt line to free

void *dev_id
Device identity to free

Remove an interrupt handler. The handler is removed and if the interrupt line is no longer in use by any driver it is disabled. On a shared IRQ the caller must ensure the interrupt is disabled on the card it drives before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.

Returns the devname argument passed to request_irq.

int **request_threaded_irq**(unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long irqflags, const char *devname, void *dev_id)

allocate an interrupt line

Parameters

unsigned int irq
Interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs. Primary handler for threaded interrupts. If handler is NULL and thread_fn != NULL the default primary handler is installed.

irq_handler_t thread_fn
Function called from the irq handler thread If NULL, no irq thread is created

unsigned long irqflags
Interrupt type flags

const char *devname
An ascii name for the claiming device

void *dev_id
A cookie passed back to the handler function

This call allocates interrupt resources and enables the interrupt line and IRQ handling. From the point this call is made your handler function may be invoked. Since your handler function must clear any interrupt the board raises, you must take care both to initialise your hardware and to set up the interrupt handler in the right order.

If you want to set up a threaded irq handler for your device then you need to supply **handler** and **thread_fn**. **handler** is still called in hard interrupt context and has to check whether the interrupt originates from the device. If yes it needs to disable the interrupt on the device and return IRQ_WAKE_THREAD which will wake up the handler thread and run **thread_fn**. This split handler design is necessary to support shared interrupts.

Dev_id must be globally unique. Normally the address of the device data structure is used as the cookie. Since the handler receives this value it makes sense to use it.

If your interrupt is shared you must pass a non NULL dev_id as this is required when freeing the interrupt.

Flags:

IRQF_SHARED Interrupt is shared IRQF_TRIGGER_* Specify active edge(s) or level IRQF_ONESHOT Run thread_fn with interrupt line masked

int **request_any_context_irq**(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id)

allocate an interrupt line

Parameters

unsigned int irq
Interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs. Threaded handler for threaded interrupts.

unsigned long flags
Interrupt type flags

const char *name
An ascii name for the claiming device

void *dev_id
A cookie passed back to the handler function

This call allocates interrupt resources and enables the interrupt line and IRQ handling. It selects either a hardirq or threaded handling method depending on the context.

On failure, it returns a negative value. On success, it returns either IRQC_IS_HARDIRQ or IRQC_IS_NESTED.

int **request_nmi**(unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char *name, void *dev_id)

allocate an interrupt line for NMI delivery

Parameters

unsigned int irq
Interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs. Threaded handler for threaded interrupts.

unsigned long irqflags
Interrupt type flags

const char *name
An ascii name for the claiming device

void *dev_id
A cookie passed back to the handler function

This call allocates interrupt resources and enables the interrupt line and IRQ handling. It sets up the IRQ line to be handled as an NMI.

An interrupt line delivering NMIs cannot be shared and IRQ handling cannot be threaded.

Interrupt lines requested for NMI delivering must produce per cpu interrupts and have auto enabling setting disabled.

Dev_id must be globally unique. Normally the address of the device data structure is used as the cookie. Since the handler receives this value it makes sense to use it.

If the interrupt line cannot be used to deliver NMIs, function will fail and return a negative value.

bool **irq_percpu_is_enabled**(unsigned int irq)

Check whether the per cpu irq is enabled

Parameters

unsigned int irq
Linux irq number to check for

Description

Must be called from a non migratable context. Returns the enable state of a per cpu interrupt on the current cpu.

void **remove_percpu_irq**(unsigned int irq, struct *irqaction* *act)

free a per-cpu interrupt

Parameters

unsigned int irq
Interrupt line to free

struct irqaction *act
irqaction for the interrupt

Description

Used to remove interrupts statically setup by the early boot process.

void **free_percpu_irq**(unsigned int irq, void __percpu *dev_id)

free an interrupt allocated with request_percpu_irq

Parameters

unsigned int irq
Interrupt line to free

void __percpu *dev_id
Device identity to free

Remove a percpu interrupt handler. The handler is removed, but the interrupt line is not disabled. This must be done on each CPU before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.

int **setup_percpu_irq**(unsigned int irq, struct *irqaction* *act)

setup a per-cpu interrupt

Parameters

unsigned int irq
Interrupt line to setup

struct irqaction *act
irqaction for the interrupt

Description

Used to statically setup per-cpu interrupts in the early boot process.

int **__request_percpu_irq**(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *devname, void __percpu *dev_id)

allocate a percpu interrupt line

Parameters

unsigned int irq
Interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs.

unsigned long flags
Interrupt type flags (IRQF_TIMER only)

const char *devname
An ascii name for the claiming device

void __percpu *dev_id
A percpu cookie passed back to the handler function

This call allocates interrupt resources and enables the interrupt on the local CPU. If the interrupt is supposed to be enabled on other CPUs, it has to be done on each CPU using enable_percpu_irq().

Dev_id must be globally unique. It is a per-cpu variable, and the handler gets called with the interrupted CPU's instance of that variable.

int **request_percpu_nmi**(unsigned int irq, irq_handler_t handler, const char *name, void __percpu *dev_id)

allocate a percpu interrupt line for NMI delivery

Parameters

unsigned int irq
Interrupt line to allocate

irq_handler_t handler
Function to be called when the IRQ occurs.

const char *name
An ascii name for the claiming device

void __percpu *dev_id
A percpu cookie passed back to the handler function

This call allocates interrupt resources for a per CPU NMI. Per CPU NMIs have to be setup on each CPU by calling *prepare_percpu_nmi()* before being enabled on the same CPU by using enable_percpu_nmi().

Dev_id must be globally unique. It is a per-cpu variable, and the handler gets called with the interrupted CPU's instance of that variable.

Interrupt lines requested for NMI delivering should have auto enabling setting disabled.

If the interrupt line cannot be used to deliver NMIs, function will fail returning a negative value.

int **prepare_percpu_nmi**(unsigned int irq)

performs CPU local setup for NMI delivery

Parameters

unsigned int irq
Interrupt line to prepare for NMI delivery

This call prepares an interrupt line to deliver NMI on the current CPU, before that interrupt line gets enabled with enable_percpu_nmi().

As a CPU local operation, this should be called from non-preemptible context.

If the interrupt line cannot be used to deliver NMIs, function will fail returning a negative value.

void **teardown_percpu_nmi**(unsigned int irq)
undoes NMI setup of IRQ line

Parameters

unsigned int irq
Interrupt line from which CPU local NMI configuration should be removed

This call undoes the setup done by *prepare_percpu_nmi()*.

IRQ line should not be enabled for the current CPU.

As a CPU local operation, this should be called from non-preemptible context.

int **irq_get_irqchip_state**(unsigned int irq, enum irqchip_irq_state which, bool *state)
returns the irqchip state of a interrupt.

Parameters

unsigned int irq
Interrupt line that is forwarded to a VM

enum irqchip_irq_state which
One of IRQCHIP_STATE_* the caller wants to know about

bool *state
a pointer to a boolean where the state is to be stored

This call snapshots the internal irqchip state of an interrupt, returning into **state** the bit corresponding to stage **which**

This function should be called with preemption disabled if the interrupt controller has per-cpu registers.

int **irq_set_irqchip_state**(unsigned int irq, enum irqchip_irq_state which, bool val)
set the state of a forwarded interrupt.

Parameters

unsigned int irq
Interrupt line that is forwarded to a VM

enum irqchip_irq_state which
State to be restored (one of IRQCHIP_STATE_*)

bool val
Value corresponding to **which**

This call sets the internal irqchip state of an interrupt, depending on the value of **which**.

This function should be called with migration disabled if the interrupt controller has per-cpu registers.

bool **irq_has_action**(unsigned int irq)

Check whether an interrupt is requested

Parameters

unsigned int irq
The linux irq number

Return

A snapshot of the current state

bool **irq_check_status_bit**(unsigned int irq, unsigned int bitmask)

Check whether bits in the irq descriptor status are set

Parameters

unsigned int irq
The linux irq number

unsigned int bitmask
The bitmask to evaluate

Return

True if one of the bits in **bitmask** is set

int **irq_set_chip**(unsigned int irq, const struct *irq_chip* *chip)

set the irq chip for an irq

Parameters

unsigned int irq
irq number

const struct irq_chip *chip
pointer to irq chip description structure

int **irq_set_irq_type**(unsigned int irq, unsigned int type)

set the irq trigger type for an irq

Parameters

unsigned int irq
irq number

unsigned int type
IRQ_TYPE_{LEVEL,EDGE}_* value - see include/linux/irq.h

int **irq_set_handler_data**(unsigned int irq, void *data)

set irq handler data for an irq

Parameters

unsigned int irq
Interrupt number

void *data
Pointer to interrupt specific data

Set the hardware irq controller data for an irq

int **irq_set_chip_data**(unsigned int irq, void *data)
set irq chip data for an irq

Parameters

unsigned int irq
Interrupt number

void *data
Pointer to chip specific data

Set the hardware irq chip data for an irq

void **handle_simple_irq**(struct irq_desc *desc)
Simple and software-decoded IRQs.

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Simple interrupts are either sent from a demultiplexing interrupt handler or come from hardware, where no interrupt hardware control is necessary.

Note

The caller is expected to handle the ack, clear, mask and
unmask issues if necessary.

void **handle_untracked_irq**(struct irq_desc *desc)
Simple and software-decoded IRQs.

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Untracked interrupts are sent from a demultiplexing interrupt handler when the demultiplexer does not know which device it its multiplexed irq domain generated the interrupt. IRQ's handled through here are not subjected to stats tracking, randomness, or spurious interrupt detection.

Note

Like handle_simple_irq, the caller is expected to handle
the ack, clear, mask and unmask issues if necessary.

void **handle_level_irq**(struct irq_desc *desc)
Level type irq handler

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Level type interrupts are active as long as the hardware line has the active level. This may require to mask the interrupt and unmask it after the associated handler has acknowledged the device, so the interrupt line is back to inactive.

void **handle_fasteoi_irq**(struct irq_desc *desc)

irq handler for transparent controllers

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Only a single callback will be issued to the chip: an ->eoi() call when the interrupt has been serviced. This enables support for modern forms of interrupt handlers, which handle the flow details in hardware, transparently.

void **handle_fasteoi_nmi**(struct irq_desc *desc)

irq handler for NMI interrupt lines

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

A simple NMI-safe handler, considering the restrictions from request_nmi.

Only a single callback will be issued to the chip: an ->eoi() call when the interrupt has been serviced. This enables support for modern forms of interrupt handlers, which handle the flow details in hardware, transparently.

void **handle_edge_irq**(struct irq_desc *desc)

edge type IRQ handler

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Interrupt occurs on the falling and/or rising edge of a hardware signal. The occurrence is latched into the irq controller hardware and must be acked in order to be reenabled. After the ack another interrupt can happen on the same source even before the first one is handled by the associated event handler. If this happens it might be necessary to disable (mask) the interrupt depending on the controller hardware. This requires to reenable the interrupt inside of the loop which handles the interrupts which have arrived while the handler was running. If all pending interrupts are handled, the loop is left.

void **handle_fasteoi_ack_irq**(struct irq_desc *desc)

irq handler for edge hierarchy stacked on transparent controllers

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Like handle_fasteoi_irq(), but for use with hierarchy where the irq_chip also needs to have its ->irq_ack() function called.

void **handle_fasteoi_mask_irq**(struct irq_desc *desc)

irq handler for level hierarchy stacked on transparent controllers

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Like handle_fasteoi_irq(), but for use with hierarchy where the irq_chip also needs to have its ->irq_mask_ack() function called.

int **irq_chip_set_parent_state**(struct *irq_data* *data, enum irqchip_irq_state which, bool val)

set the state of a parent interrupt.

Parameters

struct irq_data *data
Pointer to interrupt specific data

enum irqchip_irq_state which
State to be restored (one of IRQCHIP_STATE_*)

bool val
Value corresponding to **which**

Description

Conditional success, if the underlying irqchip does not implement it.

int **irq_chip_get_parent_state**(struct *irq_data* *data, enum irqchip_irq_state which, bool *state)

get the state of a parent interrupt.

Parameters

struct irq_data *data
Pointer to interrupt specific data

enum irqchip_irq_state which
one of IRQCHIP_STATE_* the caller wants to know

bool *state
a pointer to a boolean where the state is to be stored

Description

Conditional success, if the underlying irqchip does not implement it.

void **irq_chip_enable_parent**(struct *irq_data* *data)

Enable the parent interrupt (defaults to unmask if NULL)

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_disable_parent**(struct *irq_data* *data)

Disable the parent interrupt (defaults to mask if NULL)

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_ack_parent**(struct *irq_data* *data)

Acknowledge the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_mask_parent**(struct *irq_data* *data)

Mask the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_mask_ack_parent**(struct *irq_data* *data)

Mask and acknowledge the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_unmask_parent**(struct *irq_data* *data)

Unmask the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_eoi_parent**(struct *irq_data* *data)

Invoke EOI on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

int **irq_chip_set_affinity_parent**(struct *irq_data* *data, const struct cpumask *dest, bool force)

Set affinity on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

const struct cpumask *dest
The affinity mask to set

bool force
Flag to enforce setting (disable online checks)

Description

Conditional, as the underlying parent chip might not implement it.

int **irq_chip_set_type_parent**(struct *irq_data* *data, unsigned int type)
Set IRQ type on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

unsigned int type
IRQ_TYPE_{LEVEL,EDGE}_* value - see include/linux/irq.h

Description

Conditional, as the underlying parent chip might not implement it.

int **irq_chip_retrigger_hierarchy**(struct *irq_data* *data)
Retrigger an interrupt in hardware

Parameters

struct irq_data *data
Pointer to interrupt specific data

Description

Iterate through the domain hierarchy of the interrupt and check whether a hw retrigger function exists. If yes, invoke it.

int **irq_chip_set_vcpu_affinity_parent**(struct *irq_data* *data, void *vcpu_info)
Set vcpu affinity on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void *vcpu_info
The vcpu affinity information

int **irq_chip_set_wake_parent**(struct *irq_data* *data, unsigned int on)
Set/reset wake-up on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

unsigned int on
Whether to set or reset the wake-up capability of this irq

Description

Conditional, as the underlying parent chip might not implement it.

int **irq_chip_request_resources_parent**(struct *irq_data* *data)
Request resources on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

void **irq_chip_release_resources_parent**(struct *irq_data* *data)
Release resources on the parent interrupt

Parameters

struct irq_data *data
Pointer to interrupt specific data

5.4.10 Internal Functions Provided

This chapter contains the autogenerated documentation of the internal functions.

int **generic_handle_irq**(unsigned int irq)
Invoke the handler for a particular irq

Parameters

unsigned int irq
The irq number to handle

Return

0 on success, or -EINVAL if conversion has failed

This function must be called from an IRQ context with irq regs initialized.

int **generic_handle_irq_safe**(unsigned int irq)
Invoke the handler for a particular irq from any context.

Parameters

unsigned int irq
The irq number to handle

Return

0 on success, a negative value on error.

Description

This function can be called from any context (IRQ or process context). It will report an error if not invoked from IRQ context and the irq has been marked to enforce IRQ-context only.

int **generic_handle_domain_irq**(struct irq_domain *domain, unsigned int hwirq)
Invoke the handler for a HW irq belonging to a domain.

Parameters

struct irq_domain *domain
The domain where to perform the lookup

unsigned int hwirq
The HW irq number to convert to a logical one

Return

0 on success, or -EINVAL if conversion has failed

This function must be called from an IRQ context with irq regs initialized.

int **generic_handle_domain_nmi**(struct irq_domain *domain, unsigned int hwirq)
Invoke the handler for a HW nmi belonging to a domain.

Parameters

struct irq_domain *domain
The domain where to perform the lookup

unsigned int hwirq
The HW irq number to convert to a logical one

Return

0 on success, or -EINVAL if conversion has failed

This function must be called from an NMI context with irq regs initialized.

void **irq_free_descs**(unsigned int from, unsigned int cnt)
free irq descriptors

Parameters

unsigned int from
Start of descriptor range

unsigned int cnt
Number of consecutive irqs to free

int __ref **__irq_alloc_descs**(int irq, unsigned int from, unsigned int cnt, int node, struct module *owner, const struct *irq_affinity_desc* *affinity)
allocate and initialize a range of irq descriptors

Parameters

int irq
Allocate for specific irq number if irq >= 0

unsigned int from
Start the search from this irq number

unsigned int cnt
Number of consecutive irqs to allocate.

int node
Preferred node on which the irq descriptor should be allocated

struct module *owner
Owning module (can be NULL)

const struct irq_affinity_desc *affinity
Optional pointer to an affinity mask array of size **cnt** which hints where the irq descriptors should be allocated and which default affinities to use

Description

Returns the first irq number or error code

unsigned int **irq_get_next_irq**(unsigned int offset)
get next allocated irq number

Parameters

unsigned int offset
where to start the search

Description

Returns next irq number after offset or nr_irqs if none is found.

unsigned int **kstat_irqs_cpu**(unsigned int irq, int cpu)
Get the statistics for an interrupt on a cpu

Parameters

unsigned int irq
The interrupt number

int cpu
The cpu number

Description

Returns the sum of interrupt counts on **cpu** since boot for **irq**. The caller must ensure that the interrupt is not removed concurrently.

unsigned int **kstat_irqs_usr**(unsigned int irq)
Get the statistics for an interrupt from thread context

Parameters

unsigned int irq
The interrupt number

Description

Returns the sum of interrupt counts on all cpus since boot for **irq**.

It uses rcu to protect the access since a concurrent removal of an interrupt descriptor is observing an rcu grace period before delayed_free_desc()/irq_kobj_release().

void **handle_bad_irq**(struct irq_desc *desc)
handle spurious and unhandled irqs

Parameters

struct irq_desc *desc
description of the interrupt

Description

Handles spurious and unhandled IRQ's. It also prints a debugmessage.

void noinstr **generic_handle_arch_irq**(struct pt_regs *regs)
root irq handler for architectures which do no entry accounting themselves

Parameters

struct pt_regs *regs
Register file coming from the low-level handling code

int **irq_set_msi_desc_off**(unsigned int irq_base, unsigned int irq_offset, struct msi_desc *entry)
set MSI descriptor data for an irq at offset

Parameters

unsigned int irq_base
Interrupt number base

unsigned int irq_offset
Interrupt number offset

struct msi_desc *entry
Pointer to MSI descriptor data

Set the MSI descriptor entry for an irq at offset

int **irq_set_msi_desc**(unsigned int irq, struct msi_desc *entry)
set MSI descriptor data for an irq

Parameters

unsigned int irq
Interrupt number

struct msi_desc *entry
Pointer to MSI descriptor data

Set the MSI descriptor entry for an irq

void **irq_disable**(struct irq_desc *desc)
Mark interrupt disabled

Parameters

struct irq_desc *desc
irq descriptor which should be disabled

Description

If the chip does not implement the irq_disable callback, we use a lazy disable approach. That means we mark the interrupt disabled, but leave the hardware unmasked. That's an optimization because we avoid the hardware access for the common case where no interrupt happens after we marked it disabled. If an interrupt happens, then the interrupt flow handler masks the line at the hardware level and marks it pending.

If the interrupt chip does not implement the irq_disable callback, a driver can disable the lazy approach for a particular irq line by calling 'irq_set_status_flags(irq, IRQ_DISABLE_UNLAZY)'. This can be used for devices which cannot disable the interrupt at the device level under certain circumstances and have to use disable_irq[_nosync] instead.

void **handle_edge_eoi_irq**(struct irq_desc *desc)
edge eoi type IRQ handler

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Description

Similar as the above handle_edge_irq, but using eoi and w/o the mask/unmask logic.

void **handle_percpu_irq**(struct irq_desc *desc)
Per CPU local irq handler

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Per CPU interrupts on SMP machines without locking requirements

void **handle_percpu_devid_irq**(struct irq_desc *desc)
Per CPU local irq handler with per cpu dev ids

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Description

Per CPU interrupts on SMP machines without locking requirements. Same as handle_percpu_irq() above but with the following extras:

action->percpu_dev_id is a pointer to percpu variables which contain the real device id for the cpu on which this handler is called

void **handle_percpu_devid_fasteoi_nmi**(struct irq_desc *desc)
Per CPU local NMI handler with per cpu dev ids

Parameters

struct irq_desc *desc
the interrupt description structure for this irq

Description

Similar to handle_fasteoi_nmi, but handling the dev_id cookie as a percpu pointer.

void **irq_cpu_online**(void)
Invoke all irq_cpu_online functions.

Parameters

void
no arguments

Description

Iterate through all irqs and invoke the chip.*irq_cpu_online()* for each.

void **irq_cpu_offline**(void)
Invoke all irq_cpu_offline functions.

Parameters

void
no arguments

Description

Iterate through all irqs and invoke the chip.*irq_cpu_offline()* for each.

int **irq_chip_compose_msi_msg**(struct *irq_data* *data, struct msi_msg *msg)
Compose msi message for a irq chip

Parameters

struct irq_data *data
Pointer to interrupt specific data

struct msi_msg *msg
Pointer to the MSI message

Description

For hierarchical domains we find the first chip in the hierarchy which implements the irq_compose_msi_msg callback. For non hierarchical we use the top level chip.

int **irq_chip_pm_get**(struct *irq_data* *data)
Enable power for an IRQ chip

Parameters

struct irq_data *data
Pointer to interrupt specific data

Description

Enable the power to the IRQ chip referenced by the interrupt data structure.

int **irq_chip_pm_put**(struct *irq_data* *data)
Disable power for an IRQ chip

Parameters

struct irq_data *data
Pointer to interrupt specific data

Description

Disable the power to the IRQ chip referenced by the interrupt data structure, belongs. Note that power will only be disabled, once this function has been called for all IRQs that have called *irq_chip_pm_get()*.

5.4.11 Credits

The following people have contributed to this document:

1. Thomas Gleixner tglx@linutronix.de
2. Ingo Molnar mingo@elte.hu

5.5 Memory Protection Keys

Memory Protection Keys provide a mechanism for enforcing page-based protections, but without requiring modification of the page tables when an application changes protection domains.

Pkeys Userspace (PKU) is a feature which can be found on:

- Intel server CPUs, Skylake and later
- Intel client CPUs, Tiger Lake (11th Gen Core) and later
- Future AMD CPUs

Pkeys work by dedicating 4 previously Reserved bits in each page table entry to a "protection key", giving 16 possible keys.

Protections for each key are defined with a per-CPU user-accessible register (PKRU). Each of these is a 32-bit register storing two bits (Access Disable and Write Disable) for each of 16 keys.

Being a CPU register, PKRU is inherently thread-local, potentially giving each thread a different set of protections from every other thread.

There are two instructions (RDPKRU/WRPKRU) for reading and writing to the register. The feature is only available in 64-bit mode, even though there is theoretically space in the PAE PTEs. These permissions are enforced on data access only and have no effect on instruction fetches.

5.5.1 Syscalls

There are 3 system calls which directly interact with pkeys:

```
int pkey_alloc(unsigned long flags, unsigned long init_access_rights)
int pkey_free(int pkey);
int pkey_mprotect(unsigned long start, size_t len,
                  unsigned long prot, int pkey);
```

Before a pkey can be used, it must first be allocated with pkey_alloc(). An application calls the WRPKRU instruction directly in order to change access permissions to memory covered with a key. In this example WRPKRU is wrapped by a C function called pkey_set().

```
int real_prot = PROT_READ|PROT_WRITE;
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
ptr = mmap(NULL, PAGE_SIZE, PROT_NONE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
ret = pkey_mprotect(ptr, PAGE_SIZE, real_prot, pkey);
... application runs here
```

Now, if the application needs to update the data at 'ptr', it can gain access, do the update, then remove its write access:

```
pkey_set(pkey, 0); // clear PKEY_DISABLE_WRITE
*ptr = foo; // assign something
pkey_set(pkey, PKEY_DISABLE_WRITE); // set PKEY_DISABLE_WRITE again
```

Now when it frees the memory, it will also free the pkey since it is no longer in use:

```
munmap(ptr, PAGE_SIZE);
pkey_free(pkey);
```

Note: pkey_set() is a wrapper for the RDPKRU and WRPKRU instructions. An example implementation can be found in tools/testing/selftests/x86/protection_keys.c.

5.5.2 Behavior

The kernel attempts to make protection keys consistent with the behavior of a plain mprotect(). For instance if you do this:

```
mprotect(ptr, size, PROT_NONE);
something(ptr);
```

you can expect the same effects with protection keys when doing this:

```
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE | PKEY_DISABLE_READ);
pkey_mprotect(ptr, size, PROT_READ|PROT_WRITE, pkey);
something(ptr);
```

That should be true whether something() is a direct access to 'ptr' like:

```
*ptr = foo;
```

or when the kernel does the access on the application's behalf like with a read():

```
read(fd, ptr, 1);
```

The kernel will send a SIGSEGV in both cases, but si_code will be set to SEGV_PKERR when violating protection keys versus SEGV_ACCERR when the plain mprotect() permissions are violated.

CHAPTER

SIX

MEMORY MANAGEMENT

How to allocate and use memory in the kernel. Note that there is a lot more memory-management documentation in Documentation/mm/index.rst.

6.1 Memory Allocation Guide

Linux provides a variety of APIs for memory allocation. You can allocate small chunks using *kmalloc* or *kmem_cache_alloc* families, large virtually contiguous areas using *vmalloc* and its derivatives, or you can directly request pages from the page allocator with *alloc_pages*. It is also possible to use more specialized allocators, for instance *cma_alloc* or *zs_malloc*.

Most of the memory allocation APIs use GFP flags to express how that memory should be allocated. The GFP acronym stands for "get free pages", the underlying memory allocation function.

Diversity of the allocation APIs combined with the numerous GFP flags makes the question "How should I allocate memory?" not that easy to answer, although very likely you should use

```
kzalloc(<size>, GFP_KERNEL);
```

Of course there are cases when other allocation APIs and different GFP flags must be used.

6.1.1 Get Free Page flags

The GFP flags control the allocators behavior. They tell what memory zones can be used, how hard the allocator should try to find free memory, whether the memory can be accessed by the userspace etc. The *Documentation/core-api/mm-api.rst* provides reference documentation for the GFP flags and their combinations and here we briefly outline their recommended usage:

- Most of the time `GFP_KERNEL` is what you need. Memory for the kernel data structures, DMAable memory, inode cache, all these and many other allocations types can use `GFP_KERNEL`. Note, that using `GFP_KERNEL` implies `GFP_RECLAIM`, which means that direct reclaim may be triggered under memory pressure; the calling context must be allowed to sleep.
- If the allocation is performed from an atomic context, e.g interrupt handler, use `GFP_NOWAIT`. This flag prevents direct reclaim and IO or filesystem operations. Consequently, under memory pressure `GFP_NOWAIT` allocation is likely to fail. Allocations which have a reasonable fallback should be using `GFP_NOWARN`.

- If you think that accessing memory reserves is justified and the kernel will be stressed unless allocation succeeds, you may use `GFP_ATOMIC`.
- Untrusted allocations triggered from userspace should be a subject of kmem accounting and must have `__GFP_ACCOUNT` bit set. There is the handy `GFP_KERNEL_ACCOUNT` shortcut for `GFP_KERNEL` allocations that should be accounted.
- Userspace allocations should use either of the `GFP_USER`, `GFP_HIGHUSER` or `GFP_HIGHUSER_MOVABLE` flags. The longer the flag name the less restrictive it is.

  `GFP_HIGHUSER_MOVABLE` does not require that allocated memory will be directly accessible by the kernel and implies that the data is movable.

  `GFP_HIGHUSER` means that the allocated memory is not movable, but it is not required to be directly accessible by the kernel. An example may be a hardware allocation that maps data directly into userspace but has no addressing limitations.

  `GFP_USER` means that the allocated memory is not movable and it must be directly accessible by the kernel.

You may notice that quite a few allocations in the existing code specify `GFP_NOIO` or `GFP_NOFS`. Historically, they were used to prevent recursion deadlocks caused by direct memory reclaim calling back into the FS or IO paths and blocking on already held resources. Since 4.12 the preferred way to address this issue is to use new scope APIs described in *Documentation/core-api/gfp_mask-from-fs-io.rst*.

Other legacy GFP flags are `GFP_DMA` and `GFP_DMA32`. They are used to ensure that the allocated memory is accessible by hardware with limited addressing capabilities. So unless you are writing a driver for a device with such restrictions, avoid using these flags. And even with hardware with restrictions it is preferable to use *dma_alloc** APIs.

GFP flags and reclaim behavior

Memory allocations may trigger direct or background reclaim and it is useful to understand how hard the page allocator will try to satisfy that or another request.

- `GFP_KERNEL & ~__GFP_RECLAIM` - optimistic allocation without _any_ attempt to free memory at all. The most light weight mode which even doesn't kick the background reclaim. Should be used carefully because it might deplete the memory and the next user might hit the more aggressive reclaim.
- `GFP_KERNEL & ~__GFP_DIRECT_RECLAIM` (or `GFP_NOWAIT`)- optimistic allocation without any attempt to free memory from the current context but can wake kswapd to reclaim memory if the zone is below the low watermark. Can be used from either atomic contexts or when the request is a performance optimization and there is another fallback for a slow path.
- `(GFP_KERNEL|__GFP_HIGH) & ~__GFP_DIRECT_RECLAIM` (aka `GFP_ATOMIC`) - non sleeping allocation with an expensive fallback so it can access some portion of memory reserves. Usually used from interrupt/bottom-half context with an expensive slow path fallback.
- `GFP_KERNEL` - both background and direct reclaim are allowed and the **default** page allocator behavior is used. That means that not costly allocation requests are basically no-fail but there is no guarantee of that behavior so failures have to be checked properly by callers (e.g. OOM killer victim is allowed to fail currently).

- `GFP_KERNEL | __GFP_NORETRY` - overrides the default allocator behavior and all allocation requests fail early rather than cause disruptive reclaim (one round of reclaim in this implementation). The OOM killer is not invoked.
- `GFP_KERNEL | __GFP_RETRY_MAYFAIL` - overrides the default allocator behavior and all allocation requests try really hard. The request will fail if the reclaim cannot make any progress. The OOM killer won't be triggered.
- `GFP_KERNEL | __GFP_NOFAIL` - overrides the default allocator behavior and all allocation requests will loop endlessly until they succeed. This might be really dangerous especially for larger orders.

6.1.2 Selecting memory allocator

The most straightforward way to allocate memory is to use a function from the kmalloc() family. And, to be on the safe side it's best to use routines that set memory to zero, like kzalloc(). If you need to allocate memory for an array, there are kmalloc_array() and kcalloc() helpers. The helpers struct_size(), array_size() and array3_size() can be used to safely calculate object sizes without overflowing.

The maximal size of a chunk that can be allocated with *kmalloc* is limited. The actual limit depends on the hardware and the kernel configuration, but it is a good practice to use *kmalloc* for objects smaller than page size.

The address of a chunk allocated with *kmalloc* is aligned to at least ARCH_KMALLOC_MINALIGN bytes. For sizes which are a power of two, the alignment is also guaranteed to be at least the respective size.

Chunks allocated with kmalloc() can be resized with krealloc(). Similarly to kmalloc_array(): a helper for resizing arrays is provided in the form of krealloc_array().

For large allocations you can use vmalloc() and vzalloc(), or directly request pages from the page allocator. The memory allocated by *vmalloc* and related functions is not physically contiguous.

If you are not sure whether the allocation size is too large for *kmalloc*, it is possible to use kvmalloc() and its derivatives. It will try to allocate memory with *kmalloc* and if the allocation fails it will be retried with *vmalloc*. There are restrictions on which GFP flags can be used with *kvmalloc*; please see kvmalloc_node() reference documentation. Note that *kvmalloc* may return memory that is not physically contiguous.

If you need to allocate many identical objects you can use the slab cache allocator. The cache should be set up with kmem_cache_create() or kmem_cache_create_usercopy() before it can be used. The second function should be used if a part of the cache might be copied to the userspace. After the cache is created kmem_cache_alloc() and its convenience wrappers can allocate memory from that cache.

When the allocated memory is no longer needed it must be freed.

Objects allocated by *kmalloc* can be freed by *kfree* or *kvfree*. Objects allocated by *kmem_cache_alloc* can be freed with *kmem_cache_free*, *kfree* or *kvfree*, where the latter two might be more convenient thanks to not needing the kmem_cache pointer.

The same rules apply to _bulk and _rcu flavors of freeing functions.

Memory allocated by *vmalloc* can be freed with *vfree* or *kvfree*. Memory allocated by *kvmalloc* can be freed with *kvfree*. Caches created by *kmem_cache_create* should be freed with *kmem_cache_destroy* only after freeing all the allocated objects first.

6.2 Unaligned Memory Accesses

Author
Daniel Drake <dsd@gentoo.org>,

Author
Johannes Berg <johannes@sipsolutions.net>

With help from
Alan Cox, Avuton Olrich, Heikki Orsila, Jan Engelhardt, Kyle McMartin, Kyle Moffett, Randy Dunlap, Robert Hancock, Uli Kunitz, Vadim Lobanov

Linux runs on a wide variety of architectures which have varying behaviour when it comes to memory access. This document presents some details about unaligned accesses, why you need to write code that doesn't cause them, and how to write such code!

6.2.1 The definition of an unaligned access

Unaligned memory accesses occur when you try to read N bytes of data starting from an address that is not evenly divisible by N (i.e. addr % N != 0). For example, reading 4 bytes of data from address 0x10004 is fine, but reading 4 bytes of data from address 0x10005 would be an unaligned memory access.

The above may seem a little vague, as memory access can happen in different ways. The context here is at the machine code level: certain instructions read or write a number of bytes to or from memory (e.g. movb, movw, movl in x86 assembly). As will become clear, it is relatively easy to spot C statements which will compile to multiple-byte memory access instructions, namely when dealing with types such as u16, u32 and u64.

6.2.2 Natural alignment

The rule mentioned above forms what we refer to as natural alignment: When accessing N bytes of memory, the base memory address must be evenly divisible by N, i.e. addr % N == 0.

When writing code, assume the target architecture has natural alignment requirements.

In reality, only a few architectures require natural alignment on all sizes of memory access. However, we must consider ALL supported architectures; writing code that satisfies natural alignment requirements is the easiest way to achieve full portability.

6.2.3 Why unaligned access is bad

The effects of performing an unaligned memory access vary from architecture to architecture. It would be easy to write a whole document on the differences here; a summary of the common scenarios is presented below:

- Some architectures are able to perform unaligned memory accesses transparently, but there is usually a significant performance cost.
- Some architectures raise processor exceptions when unaligned accesses happen. The exception handler is able to correct the unaligned access, at significant cost to performance.

- Some architectures raise processor exceptions when unaligned accesses happen, but the exceptions do not contain enough information for the unaligned access to be corrected.
- Some architectures are not capable of unaligned memory access, but will silently perform a different memory access to the one that was requested, resulting in a subtle code bug that is hard to detect!

It should be obvious from the above that if your code causes unaligned memory accesses to happen, your code will not work correctly on certain platforms and will cause performance problems on others.

6.2.4 Code that does not cause unaligned access

At first, the concepts above may seem a little hard to relate to actual coding practice. After all, you don't have a great deal of control over memory addresses of certain variables, etc.

Fortunately things are not too complex, as in most cases, the compiler ensures that things will work for you. For example, take the following structure:

```
struct foo {
        u16 field1;
        u32 field2;
        u8 field3;
};
```

Let us assume that an instance of the above structure resides in memory starting at address 0x10000. With a basic level of understanding, it would not be unreasonable to expect that accessing field2 would cause an unaligned access. You'd be expecting field2 to be located at offset 2 bytes into the structure, i.e. address 0x10002, but that address is not evenly divisible by 4 (remember, we're reading a 4 byte value here).

Fortunately, the compiler understands the alignment constraints, so in the above case it would insert 2 bytes of padding in between field1 and field2. Therefore, for standard structure types you can always rely on the compiler to pad structures so that accesses to fields are suitably aligned (assuming you do not cast the field to a type of different length).

Similarly, you can also rely on the compiler to align variables and function parameters to a naturally aligned scheme, based on the size of the type of the variable.

At this point, it should be clear that accessing a single byte (u8 or char) will never cause an unaligned access, because all memory addresses are evenly divisible by one.

On a related topic, with the above considerations in mind you may observe that you could reorder the fields in the structure in order to place fields where padding would otherwise be inserted, and hence reduce the overall resident memory size of structure instances. The optimal layout of the above example is:

```
struct foo {
        u32 field2;
        u16 field1;
        u8 field3;
};
```

For a natural alignment scheme, the compiler would only have to add a single byte of padding at the end of the structure. This padding is added in order to satisfy alignment constraints for arrays of these structures.

Another point worth mentioning is the use of __attribute__((packed)) on a structure type. This GCC-specific attribute tells the compiler never to insert any padding within structures, useful when you want to use a C struct to represent some data that comes in a fixed arrangement 'off the wire'.

You might be inclined to believe that usage of this attribute can easily lead to unaligned accesses when accessing fields that do not satisfy architectural alignment requirements. However, again, the compiler is aware of the alignment constraints and will generate extra instructions to perform the memory access in a way that does not cause unaligned access. Of course, the extra instructions obviously cause a loss in performance compared to the non-packed case, so the packed attribute should only be used when avoiding structure padding is of importance.

6.2.5 Code that causes unaligned access

With the above in mind, let's move onto a real life example of a function that can cause an unaligned memory access. The following function taken from include/linux/etherdevice.h is an optimized routine to compare two ethernet MAC addresses for equality:

```
bool ether_addr_equal(const u8 *addr1, const u8 *addr2)
{
#ifdef CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
      u32 fold = ((*(const u32 *)addr1) ^ (*(const u32 *)addr2)) |
                 ((*(const u16 *)(addr1 + 4)) ^ (*(const u16 *)(addr2 + 4)));

      return fold == 0;
#else
      const u16 *a = (const u16 *)addr1;
      const u16 *b = (const u16 *)addr2;
      return ((a[0] ^ b[0]) | (a[1] ^ b[1]) | (a[2] ^ b[2])) == 0;
#endif
}
```

In the above function, when the hardware has efficient unaligned access capability, there is no issue with this code. But when the hardware isn't able to access memory on arbitrary boundaries, the reference to a[0] causes 2 bytes (16 bits) to be read from memory starting at address addr1.

Think about what would happen if addr1 was an odd address such as 0x10003. (Hint: it'd be an unaligned access.)

Despite the potential unaligned access problems with the above function, it is included in the kernel anyway but is understood to only work normally on 16-bit-aligned addresses. It is up to the caller to ensure this alignment or not use this function at all. This alignment-unsafe function is still useful as it is a decent optimization for the cases when you can ensure alignment, which is true almost all of the time in ethernet networking context.

Here is another example of some code that could cause unaligned accesses:

```
void myfunc(u8 *data, u32 value)
{
        [...]
        *((u32 *) data) = cpu_to_le32(value);
        [...]
}
```

This code will cause unaligned accesses every time the data parameter points to an address that is not evenly divisible by 4.

In summary, the 2 main scenarios where you may run into unaligned access problems involve:

1. Casting variables to types of different lengths
2. Pointer arithmetic followed by access to at least 2 bytes of data

6.2.6 Avoiding unaligned accesses

The easiest way to avoid unaligned access is to use the get_unaligned() and put_unaligned() macros provided by the <asm/unaligned.h> header file.

Going back to an earlier example of code that potentially causes unaligned access:

```
void myfunc(u8 *data, u32 value)
{
        [...]
        *((u32 *) data) = cpu_to_le32(value);
        [...]
}
```

To avoid the unaligned memory access, you would rewrite it as follows:

```
void myfunc(u8 *data, u32 value)
{
        [...]
        value = cpu_to_le32(value);
        put_unaligned(value, (u32 *) data);
        [...]
}
```

The get_unaligned() macro works similarly. Assuming 'data' is a pointer to memory and you wish to avoid unaligned access, its usage is as follows:

```
u32 value = get_unaligned((u32 *) data);
```

These macros work for memory accesses of any length (not just 32 bits as in the examples above). Be aware that when compared to standard access of aligned memory, using these macros to access unaligned memory can be costly in terms of performance.

If use of such macros is not convenient, another option is to use memcpy(), where the source or destination (or both) are of type u8* or unsigned char*. Due to the byte-wise nature of this operation, unaligned accesses are avoided.

6.2.7 Alignment vs. Networking

On architectures that require aligned loads, networking requires that the IP header is aligned on a four-byte boundary to optimise the IP stack. For regular ethernet hardware, the constant NET_IP_ALIGN is used. On most architectures this constant has the value 2 because the normal ethernet header is 14 bytes long, so in order to get proper alignment one needs to DMA to an address which can be expressed as 4*n + 2. One notable exception here is powerpc which defines NET_IP_ALIGN to 0 because DMA to unaligned addresses can be very expensive and dwarf the cost of unaligned loads.

For some ethernet hardware that cannot DMA to unaligned addresses like 4*n+2 or non-ethernet hardware, this can be a problem, and it is then required to copy the incoming frame into an aligned buffer. Because this is unnecessary on architectures that can do unaligned accesses, the code can be made dependent on CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS like so:

```
#ifdef CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
        skb = original skb
#else
        skb = copy skb
#endif
```

6.3 Dynamic DMA mapping using the generic device

Author
James E.J. Bottomley <James.Bottomley@HansenPartnership.com>

This document describes the DMA API. For a more gentle introduction of the API (and actual examples), see *Dynamic DMA mapping Guide*.

This API is split into two pieces. Part I describes the basic API. Part II describes extensions for supporting non-consistent memory machines. Unless you know that your driver absolutely has to support non-consistent platforms (this is usually only legacy platforms) you should only use the API described in part I.

6.3.1 Part I - dma_API

To get the dma_API, you must #include <linux/dma-mapping.h>. This provides dma_addr_t and the interfaces described below.

A dma_addr_t can hold any valid DMA address for the platform. It can be given to a device to use as a DMA source or target. A CPU cannot reference a dma_addr_t directly because there may be translation between its physical address space and the DMA address space.

6.3.2 Part Ia - Using large DMA-coherent buffers

```
void *
dma_alloc_coherent(struct device *dev, size_t size,
                   dma_addr_t *dma_handle, gfp_t flag)
```

Consistent memory is memory for which a write by either the device or the processor can immediately be read by the processor or device without having to worry about caching effects. (You may however need to make sure to flush the processor's write buffers before telling devices to read that memory.)

This routine allocates a region of <size> bytes of consistent memory.

It returns a pointer to the allocated region (in the processor's virtual address space) or NULL if the allocation failed.

It also returns a <dma_handle> which may be cast to an unsigned integer the same width as the bus and given to the device as the DMA address base of the region.

Note: consistent memory can be expensive on some platforms, and the minimum allocation length may be as big as a page, so you should consolidate your requests for consistent memory as much as possible. The simplest way to do that is to use the dma_pool calls (see below).

The flag parameter (dma_alloc_coherent() only) allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation (the implementation may choose to ignore flags that affect the location of the returned memory, like GFP_DMA).

```
void
dma_free_coherent(struct device *dev, size_t size, void *cpu_addr,
                  dma_addr_t dma_handle)
```

Free a region of consistent memory you previously allocated. dev, size and dma_handle must all be the same as those passed into dma_alloc_coherent(). cpu_addr must be the virtual address returned by the dma_alloc_coherent().

Note that unlike their sibling allocation calls, these routines may only be called with IRQs enabled.

6.3.3 Part Ib - Using small DMA-coherent buffers

To get this part of the dma_API, you must #include <linux/dmapool.h>

Many drivers need lots of small DMA-coherent memory regions for DMA descriptors or I/O buffers. Rather than allocating in units of a page or more using dma_alloc_coherent(), you can use DMA pools. These work much like a struct kmem_cache, except that they use the DMA-coherent allocator, not __get_free_pages(). Also, they understand common hardware constraints for alignment, like queue heads needing to be aligned on N-byte boundaries.

```
struct dma_pool *
dma_pool_create(const char *name, struct device *dev,
                size_t size, size_t align, size_t alloc);
```

dma_pool_create() initializes a pool of DMA-coherent buffers for use with a given device. It must be called in a context which can sleep.

The "name" is for diagnostics (like a struct kmem_cache name); dev and size are like what you'd pass to dma_alloc_coherent(). The device's hardware alignment requirement for this type of data is "align" (which is expressed in bytes, and must be a power of two). If your device has no boundary crossing restrictions, pass 0 for alloc; passing 4096 says memory allocated from this pool must not cross 4KByte boundaries.

```
void *
dma_pool_zalloc(struct dma_pool *pool, gfp_t mem_flags,
                dma_addr_t *handle)
```

Wraps *dma_pool_alloc()* and also zeroes the returned memory if the allocation attempt succeeded.

```
void *
dma_pool_alloc(struct dma_pool *pool, gfp_t gfp_flags,
               dma_addr_t *dma_handle);
```

This allocates memory from the pool; the returned memory will meet the size and alignment requirements specified at creation time. Pass GFP_ATOMIC to prevent blocking, or if it's permitted (not in_interrupt, not holding SMP locks), pass GFP_KERNEL to allow blocking. Like dma_alloc_coherent(), this returns two values: an address usable by the CPU, and the DMA address usable by the pool's device.

```
void
dma_pool_free(struct dma_pool *pool, void *vaddr,
              dma_addr_t addr);
```

This puts memory back into the pool. The pool is what was passed to *dma_pool_alloc()*; the CPU (vaddr) and DMA addresses are what were returned when that routine allocated the memory being freed.

```
void
dma_pool_destroy(struct dma_pool *pool);
```

dma_pool_destroy() frees the resources of the pool. It must be called in a context which can sleep. Make sure you've freed all allocated memory back to the pool before you destroy it.

6.3.4 Part Ic - DMA addressing limitations

```
int
dma_set_mask_and_coherent(struct device *dev, u64 mask)
```

Checks to see if the mask is possible and updates the device streaming and coherent DMA mask parameters if it is.

Returns: 0 if successful and a negative error if not.

```
int
dma_set_mask(struct device *dev, u64 mask)
```

Checks to see if the mask is possible and updates the device parameters if it is.

Returns: 0 if successful and a negative error if not.

```
int
dma_set_coherent_mask(struct device *dev, u64 mask)
```

Checks to see if the mask is possible and updates the device parameters if it is.

Returns: 0 if successful and a negative error if not.

```
u64
dma_get_required_mask(struct device *dev)
```

This API returns the mask that the platform requires to operate efficiently. Usually this means the returned mask is the minimum required to cover all of memory. Examining the required mask gives drivers with variable descriptor sizes the opportunity to use smaller descriptors as necessary.

Requesting the required mask does not alter the current mask. If you wish to take advantage of it, you should issue a dma_set_mask() call to set the mask to the value returned.

```
size_t
dma_max_mapping_size(struct device *dev);
```

Returns the maximum size of a mapping for the device. The size parameter of the mapping functions like dma_map_single(), dma_map_page() and others should not be larger than the returned value.

```
size_t
dma_opt_mapping_size(struct device *dev);
```

Returns the maximum optimal size of a mapping for the device.

Mapping larger buffers may take much longer in certain scenarios. In addition, for high-rate short-lived streaming mappings, the upfront time spent on the mapping may account for an appreciable part of the total request lifetime. As such, if splitting larger requests incurs no significant performance penalty, then device drivers are advised to limit total DMA streaming mappings length to the returned value.

```
bool
dma_need_sync(struct device *dev, dma_addr_t dma_addr);
```

Returns %true if dma_sync_single_for_{device,cpu} calls are required to transfer memory ownership. Returns %false if those calls can be skipped.

```
unsigned long
dma_get_merge_boundary(struct device *dev);
```

Returns the DMA merge boundary. If the device cannot merge any the DMA address segments, the function returns 0.

6.3.5 Part Id - Streaming DMA mappings

```
dma_addr_t
dma_map_single(struct device *dev, void *cpu_addr, size_t size,
               enum dma_data_direction direction)
```

Maps a piece of processor virtual memory so it can be accessed by the device and returns the DMA address of the memory.

The direction for both APIs may be converted freely by casting. However the dma_API uses a strongly typed enumerator for its direction:

DMA_NONE	no direction (used for debugging)
DMA_TO_DEVICE	data is going from the memory to the device
DMA_FROM_DEVICE	data is coming from the device to the memory
DMA_BIDIRECTIONAL	direction isn't known

Note: Not all memory regions in a machine can be mapped by this API. Further, contiguous kernel virtual space may not be contiguous as physical memory. Since this API does not provide any scatter/gather capability, it will fail if the user tries to map a non-physically contiguous piece of memory. For this reason, memory to be mapped by this API should be obtained from sources which guarantee it to be physically contiguous (like kmalloc).

Further, the DMA address of the memory must be within the dma_mask of the device (the dma_mask is a bit mask of the addressable region for the device, i.e., if the DMA address of the memory ANDed with the dma_mask is still equal to the DMA address, then the device can perform DMA to the memory). To ensure that the memory allocated by kmalloc is within the dma_mask, the driver may specify various platform-dependent flags to restrict the DMA address range of the allocation (e.g., on x86, GFP_DMA guarantees to be within the first 16MB of available DMA addresses, as required by ISA devices).

Note also that the above constraints on physical contiguity and dma_mask may not apply if the platform has an IOMMU (a device which maps an I/O DMA address to a physical memory address). However, to be portable, device driver writers may *not* assume that such an IOMMU exists.

Warning: Memory coherency operates at a granularity called the cache line width. In order for memory mapped by this API to operate correctly, the mapped region must begin exactly on a cache line boundary and end exactly on one (to prevent two separately mapped regions from sharing a single cache line). Since the cache line size may not be known at compile time, the API will not enforce this requirement. Therefore, it is recommended that driver writers who don't take special care to determine the cache line size at run time only map virtual regions that begin and end on page boundaries (which are guaranteed also to be cache line boundaries).

DMA_TO_DEVICE synchronisation must be done after the last modification of the memory region by the software and before it is handed off to the device. Once this primitive is used, memory covered by this primitive should be treated as read-only by the device. If the device may write to it at any point, it should be DMA_BIDIRECTIONAL (see below).

DMA_FROM_DEVICE synchronisation must be done before the driver accesses data that may be changed by the device. This memory should be treated as read-only by the driver. If the driver needs to write to it at any point, it should be DMA_BIDIRECTIONAL (see below).

DMA_BIDIRECTIONAL requires special handling: it means that the driver isn't sure if the memory was modified before being handed off to the device and also isn't sure if the device will also modify it. Thus, you must always sync bidirectional memory twice: once before the memory is handed off to the device (to make sure all memory changes are flushed from the processor) and once before the data may be accessed after being used by the device (to make sure any processor cache lines are updated with data that the device may have changed).

```
void
dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                 enum dma_data_direction direction)
```

Unmaps the region previously mapped. All the parameters passed in must be identical to those passed in (and returned) by the mapping API.

```
dma_addr_t
dma_map_page(struct device *dev, struct page *page,
             unsigned long offset, size_t size,
             enum dma_data_direction direction)

void
dma_unmap_page(struct device *dev, dma_addr_t dma_address, size_t size,
               enum dma_data_direction direction)
```

API for mapping and unmapping for pages. All the notes and warnings for the other mapping APIs apply here. Also, although the <offset> and <size> parameters are provided to do partial page mapping, it is recommended that you never use these unless you really know what the cache width is.

```
dma_addr_t
dma_map_resource(struct device *dev, phys_addr_t phys_addr, size_t size,
                 enum dma_data_direction dir, unsigned long attrs)

void
dma_unmap_resource(struct device *dev, dma_addr_t addr, size_t size,
                   enum dma_data_direction dir, unsigned long attrs)
```

API for mapping and unmapping for MMIO resources. All the notes and warnings for the other mapping APIs apply here. The API should only be used to map device MMIO resources, mapping of RAM is not permitted.

```
int
dma_mapping_error(struct device *dev, dma_addr_t dma_addr)
```

In some circumstances dma_map_single(), dma_map_page() and dma_map_resource() will fail to create a mapping. A driver can check for these errors by testing the returned DMA address with dma_mapping_error(). A non-zero return value means the mapping could not be created and the driver should take appropriate action (e.g. reduce current DMA mapping usage or delay and try again later).

```
int
dma_map_sg(struct device *dev, struct scatterlist *sg,
           int nents, enum dma_data_direction direction)
```

Returns: the number of DMA address segments mapped (this may be shorter than <nents> passed in if some elements of the scatter/gather list are physically or virtually adjacent and an IOMMU maps them with a single entry).

Please note that the sg cannot be mapped again if it has been mapped once. The mapping process is allowed to destroy information in the sg.

As with the other mapping interfaces, dma_map_sg() can fail. When it does, 0 is returned and a driver must take appropriate action. It is critical that the driver do something, in the case of a block driver aborting the request or even oopsing is better than doing nothing and corrupting the filesystem.

With scatterlists, you use the resulting mapping like this:

```
int i, count = dma_map_sg(dev, sglist, nents, direction);
struct scatterlist *sg;

for_each_sg(sglist, sg, count, i) {
        hw_address[i] = sg_dma_address(sg);
        hw_len[i] = sg_dma_len(sg);
}
```

where nents is the number of entries in the sglist.

The implementation is free to merge several consecutive sglist entries into one (e.g. with an IOMMU, or if several pages just happen to be physically contiguous) and returns the actual number of sg entries it mapped them to. On failure 0, is returned.

Then you should loop count times (note: this can be less than nents times) and use sg_dma_address() and sg_dma_len() macros where you previously accessed sg->address and sg->length as shown above.

```
void
dma_unmap_sg(struct device *dev, struct scatterlist *sg,
             int nents, enum dma_data_direction direction)
```

Unmap the previously mapped scatter/gather list. All the parameters must be the same as those and passed in to the scatter/gather mapping API.

Note: <nents> must be the number you passed in, *not* the number of DMA address entries returned.

```
void
dma_sync_single_for_cpu(struct device *dev, dma_addr_t dma_handle,
                        size_t size,
                        enum dma_data_direction direction)

void
dma_sync_single_for_device(struct device *dev, dma_addr_t dma_handle,
```

```
                           size_t size,
                           enum dma_data_direction direction)

void
dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                    int nents,
                    enum dma_data_direction direction)

void
dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                       int nents,
                       enum dma_data_direction direction)
```

Synchronise a single contiguous or scatter/gather mapping for the CPU and device. With the sync_sg API, all the parameters must be the same as those passed into the single mapping API. With the sync_single API, you can use dma_handle and size parameters that aren't identical to those passed into the single mapping API to do a partial sync.

Note: You must do this:

- Before reading values that have been written by DMA from the device (use the DMA_FROM_DEVICE direction)
- After writing values that will be written to the device using DMA (use the DMA_TO_DEVICE) direction
- before *and* after handing memory to the device if the memory is DMA_BIDIRECTIONAL

See also dma_map_single().

```
dma_addr_t
dma_map_single_attrs(struct device *dev, void *cpu_addr, size_t size,
                     enum dma_data_direction dir,
                     unsigned long attrs)

void
dma_unmap_single_attrs(struct device *dev, dma_addr_t dma_addr,
                       size_t size, enum dma_data_direction dir,
                       unsigned long attrs)

int
dma_map_sg_attrs(struct device *dev, struct scatterlist *sgl,
                 int nents, enum dma_data_direction dir,
                 unsigned long attrs)

void
dma_unmap_sg_attrs(struct device *dev, struct scatterlist *sgl,
                   int nents, enum dma_data_direction dir,
                   unsigned long attrs)
```

The four functions above are just like the counterpart functions without the _attrs suffixes, except that they pass an optional dma_attrs.

The interpretation of DMA attributes is architecture-specific, and each attribute should be documented in *DMA attributes*.

If dma_attrs are 0, the semantics of each of these functions is identical to those of the corresponding function without the _attrs suffix. As a result dma_map_single_attrs() can generally replace dma_map_single(), etc.

As an example of the use of the `*_attrs` functions, here's how you could pass an attribute DMA_ATTR_FOO when mapping memory for DMA:

```
#include <linux/dma-mapping.h>
/* DMA_ATTR_FOO should be defined in linux/dma-mapping.h and
* documented in Documentation/core-api/dma-attributes.rst */
...

        unsigned long attr;
        attr |= DMA_ATTR_FOO;
        ....
        n = dma_map_sg_attrs(dev, sg, nents, DMA_TO_DEVICE, attr);
        ....
```

Architectures that care about DMA_ATTR_FOO would check for its presence in their implementations of the mapping and unmapping routines, e.g.::

```
void whizco_dma_map_sg_attrs(struct device *dev, dma_addr_t dma_addr,
                             size_t size, enum dma_data_direction dir,
                             unsigned long attrs)
{
        ....
        if (attrs & DMA_ATTR_FOO)
                /* twizzle the frobnozzle */
        ....
}
```

6.3.6 Part II - Non-coherent DMA allocations

These APIs allow to allocate pages that are guaranteed to be DMA addressable by the passed in device, but which need explicit management of memory ownership for the kernel vs the device.

If you don't understand how cache line coherency works between a processor and an I/O device, you should not be using this part of the API.

```
struct page *
dma_alloc_pages(struct device *dev, size_t size, dma_addr_t *dma_handle,
                enum dma_data_direction dir, gfp_t gfp)
```

This routine allocates a region of <size> bytes of non-coherent memory. It returns a pointer to first struct page for the region, or NULL if the allocation failed. The resulting struct page can be used for everything a struct page is suitable for.

It also returns a <dma_handle> which may be cast to an unsigned integer the same width as the bus and given to the device as the DMA address base of the region.

The dir parameter specified if data is read and/or written by the device, see dma_map_single() for details.

The gfp parameter allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation, but rejects flags used to specify a memory zone such as GFP_DMA or GFP_HIGHMEM.

Before giving the memory to the device, dma_sync_single_for_device() needs to be called, and before reading memory written by the device, dma_sync_single_for_cpu(), just like for streaming DMA mappings that are reused.

```
void
dma_free_pages(struct device *dev, size_t size, struct page *page,
                dma_addr_t dma_handle, enum dma_data_direction dir)
```

Free a region of memory previously allocated using dma_alloc_pages(). dev, size, dma_handle and dir must all be the same as those passed into dma_alloc_pages(). page must be the pointer returned by dma_alloc_pages().

```
int
dma_mmap_pages(struct device *dev, struct vm_area_struct *vma,
               size_t size, struct page *page)
```

Map an allocation returned from dma_alloc_pages() into a user address space. dev and size must be the same as those passed into dma_alloc_pages(). page must be the pointer returned by dma_alloc_pages().

```
void *
dma_alloc_noncoherent(struct device *dev, size_t size,
                dma_addr_t *dma_handle, enum dma_data_direction dir,
                gfp_t gfp)
```

This routine is a convenient wrapper around dma_alloc_pages that returns the kernel virtual address for the allocated memory instead of the page structure.

```
void
dma_free_noncoherent(struct device *dev, size_t size, void *cpu_addr,
                dma_addr_t dma_handle, enum dma_data_direction dir)
```

Free a region of memory previously allocated using dma_alloc_noncoherent(). dev, size, dma_handle and dir must all be the same as those passed into dma_alloc_noncoherent(). cpu_addr must be the virtual address returned by dma_alloc_noncoherent().

```
struct sg_table *
dma_alloc_noncontiguous(struct device *dev, size_t size,
                        enum dma_data_direction dir, gfp_t gfp,
                        unsigned long attrs);
```

This routine allocates <size> bytes of non-coherent and possibly non-contiguous memory. It returns a pointer to struct sg_table that describes the allocated and DMA mapped memory, or NULL if the allocation failed. The resulting memory can be used for struct page mapped into a scatterlist are suitable for.

The return sg_table is guaranteed to have 1 single DMA mapped segment as indicated by sgt->nents, but it might have multiple CPU side segments as indicated by sgt->orig_nents.

The dir parameter specified if data is read and/or written by the device, see dma_map_single() for details.

The gfp parameter allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation, but rejects flags used to specify a memory zone such as GFP_DMA or GFP_HIGHMEM.

The attrs argument must be either 0 or DMA_ATTR_ALLOC_SINGLE_PAGES.

Before giving the memory to the device, dma_sync_sgtable_for_device() needs to be called, and before reading memory written by the device, dma_sync_sgtable_for_cpu(), just like for streaming DMA mappings that are reused.

```
void
dma_free_noncontiguous(struct device *dev, size_t size,
                       struct sg_table *sgt,
                       enum dma_data_direction dir)
```

Free memory previously allocated using dma_alloc_noncontiguous(). dev, size, and dir must all be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

```
void *
dma_vmap_noncontiguous(struct device *dev, size_t size,
        struct sg_table *sgt)
```

Return a contiguous kernel mapping for an allocation returned from dma_alloc_noncontiguous(). dev and size must be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

Once a non-contiguous allocation is mapped using this function, the flush_kernel_vmap_range() and invalidate_kernel_vmap_range() APIs must be used to manage the coherency between the kernel mapping, the device and user space mappings (if any).

```
void
dma_vunmap_noncontiguous(struct device *dev, void *vaddr)
```

Unmap a kernel mapping returned by dma_vmap_noncontiguous(). dev must be the same the one passed into dma_alloc_noncontiguous(). vaddr must be the pointer returned by dma_vmap_noncontiguous().

```
int
dma_mmap_noncontiguous(struct device *dev, struct vm_area_struct *vma,
                       size_t size, struct sg_table *sgt)
```

Map an allocation returned from dma_alloc_noncontiguous() into a user address space. dev and size must be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

```
int
dma_get_cache_alignment(void)
```

Returns the processor cache alignment. This is the absolute minimum alignment *and* width that you must observe when either mapping memory or doing partial flushes.

Note: This API may return a number *larger* than the actual cache line, but it will guarantee that one or more cache lines fit exactly into the width returned by this call. It will also always be a power of two for easy alignment.

6.3.7 Part III - Debug drivers use of the DMA-API

The DMA-API as described above has some constraints. DMA addresses must be released with the corresponding function with the same size for example. With the advent of hardware IOMMUs it becomes more and more important that drivers do not violate those constraints. In the worst case such a violation can result in data corruption up to destroyed filesystems.

To debug drivers and find bugs in the usage of the DMA-API checking code can be compiled into the kernel which will tell the developer about those violations. If your architecture supports it you can select the "Enable debugging of DMA-API usage" option in your kernel configuration. Enabling this option has a performance impact. Do not enable it in production kernels.

If you boot the resulting kernel will contain code which does some bookkeeping about what DMA memory was allocated for which device. If this code detects an error it prints a warning message with some details into your kernel log. An example warning message may look like this:

```
WARNING: at /data2/repos/linux-2.6-iommu/lib/dma-debug.c:448
        check_unmap+0x203/0x490()
Hardware name:
forcedeth 0000:00:08.0: DMA-API: device driver frees DMA memory with wrong
        function [device address=0x00000000640444be] [size=66 bytes] [mapped as
single] [unmapped as page]
Modules linked in: nfsd exportfs bridge stp llc r8169
Pid: 0, comm: swapper Tainted: G        W  2.6.28-dmatest-09289-g8bb99c0 #1
Call Trace:
<IRQ>  [<ffffffff80240b22>] warn_slowpath+0xf2/0x130
[<ffffffff80647b70>] _spin_unlock+0x10/0x30
[<ffffffff80537e75>] usb_hcd_link_urb_to_ep+0x75/0xc0
[<ffffffff80647c22>] _spin_unlock_irqrestore+0x12/0x40
[<ffffffff8055347f>] ohci_urb_enqueue+0x19f/0x7c0
[<ffffffff80252f96>] queue_work+0x56/0x60
[<ffffffff80237e10>] enqueue_task_fair+0x20/0x50
[<ffffffff80539279>] usb_hcd_submit_urb+0x379/0xbc0
[<ffffffff803b78c3>] cpumask_next_and+0x23/0x40
[<ffffffff80235177>] find_busiest_group+0x207/0x8a0
[<ffffffff8064784f>] _spin_lock_irqsave+0x1f/0x50
[<ffffffff803c7ea3>] check_unmap+0x203/0x490
[<ffffffff803c8259>] debug_dma_unmap_page+0x49/0x50
[<ffffffff80485f26>] nv_tx_done_optimized+0xc6/0x2c0
[<ffffffff80486c13>] nv_nic_irq_optimized+0x73/0x2b0
[<ffffffff8026df84>] handle_IRQ_event+0x34/0x70
[<ffffffff8026ffe9>] handle_edge_irq+0xc9/0x150
```

```
[<ffffffff8020e3ab>] do_IRQ+0xcb/0x1c0
[<ffffffff8020c093>] ret_from_intr+0x0/0xa
<EOI> <4>---[ end trace f6435a98e2a38c0e ]---
```

The driver developer can find the driver and the device including a stacktrace of the DMA-API call which caused this warning.

Per default only the first error will result in a warning message. All other errors will only silently counted. This limitation exist to prevent the code from flooding your kernel log. To support debugging a device driver this can be disabled via debugfs. See the debugfs interface documentation below for details.

The debugfs directory for the DMA-API debugging code is called dma-api/. In this directory the following files can currently be found:

dma-api/all_errors	This file contains a numeric value. If this value is not equal to zero the debugging code will print a warning for every error it finds into the kernel log. Be careful with this option, as it can easily flood your logs.
dma-api/disabled	This read-only file contains the character 'Y' if the debugging code is disabled. This can happen when it runs out of memory or if it was disabled at boot time
dma-api/dump	This read-only file contains current DMA mappings.
dma-api/error_count	This file is read-only and shows the total numbers of errors found.
dma-api/num_errors	The number in this file shows how many warnings will be printed to the kernel log before it stops. This number is initialized to one at system boot and be set by writing into this file
dma-api/min_free_entries	This read-only file can be read to get the minimum number of free dma_debug_entries the allocator has ever seen. If this value goes down to zero the code will attempt to increase nr_total_entries to compensate.
dma-api/num_free_entries	The current number of free dma_debug_entries in the allocator.
dma-api/nr_total_entries	The total number of dma_debug_entries in the allocator, both free and used.
dma-api/driver_filter	You can write a name of a driver into this file to limit the debug output to requests from that particular driver. Write an empty string to that file to disable the filter and see all errors again.

If you have this code compiled into your kernel it will be enabled by default. If you want to boot without the bookkeeping anyway you can provide 'dma_debug=off' as a boot parameter. This will disable DMA-API debugging. Notice that you can not enable it again at runtime. You have to reboot to do so.

If you want to see debug messages only for a special device driver you can specify the dma_debug_driver=<drivername> parameter. This will enable the driver filter at boot time. The debug code will only print errors for that driver afterwards. This filter can be disabled or changed later using debugfs.

When the code disables itself at runtime this is most likely because it ran out of dma_debug_entries and was unable to allocate more on-demand. 65536 entries are preallocated at boot - if this is too low for you boot with 'dma_debug_entries=<your_desired_number>' to overwrite the default. Note that the code allocates entries in batches, so the exact number of preallocated entries may be greater than the actual number requested. The code will print to the kernel log each time it has dynamically allocated as many entries as were initially preallocated. This is to indicate that a larger preallocation size may be appropriate, or if it happens continually that a driver may be leaking mappings.

```
void
debug_dma_mapping_error(struct device *dev, dma_addr_t dma_addr);
```

dma-debug interface debug_dma_mapping_error() to debug drivers that fail to check DMA mapping errors on addresses returned by dma_map_single() and dma_map_page() interfaces. This interface clears a flag set by debug_dma_map_page() to indicate that dma_mapping_error() has been called by the driver. When driver does unmap, debug_dma_unmap() checks the flag and if this flag is still set, prints warning message that includes call trace that leads up to the unmap. This interface can be called from dma_mapping_error() routines to enable DMA mapping error check debugging.

6.4 Dynamic DMA mapping Guide

Author
David S. Miller <davem@redhat.com>

Author
Richard Henderson <rth@cygnus.com>

Author
Jakub Jelinek <jakub@redhat.com>

This is a guide to device driver writers on how to use the DMA API with example pseudo-code. For a concise description of the API, see DMA-API.txt.

6.4.1 CPU and DMA addresses

There are several kinds of addresses involved in the DMA API, and it's important to understand the differences.

The kernel normally uses virtual addresses. Any address returned by kmalloc(), vmalloc(), and similar interfaces is a virtual address and can be stored in a `void *`.

The virtual memory system (TLB, page tables, etc.) translates virtual addresses to CPU physical addresses, which are stored as "phys_addr_t" or "resource_size_t". The kernel manages device resources like registers as physical addresses. These are the addresses in /proc/iomem. The physical address is not directly useful to a driver; it must use ioremap() to map the space and produce a virtual address.

I/O devices use a third kind of address: a "bus address". If a device has registers at an MMIO address, or if it performs DMA to read or write system memory, the addresses used by the device are bus addresses. In some systems, bus addresses are identical to CPU physical addresses, but in general they are not. IOMMUs and host bridges can produce arbitrary mappings between physical and bus addresses.

From a device's point of view, DMA uses the bus address space, but it may be restricted to a subset of that space. For example, even if a system supports 64-bit addresses for main memory and PCI BARs, it may use an IOMMU so devices only need to use 32-bit DMA addresses.

Here's a picture and some examples:

```
             CPU                  CPU                  Bus
           Virtual              Physical             Address
           Address              Address               Space
            Space                Space

          +-------+             +------+             +------+
          |       |             |MMIO  |   Offset    |      |
          |       |  Virtual    |Space |   applied   |      |
        C +-------+ --------> B +------+ ----------> +------+ A
          |       |  mapping    |      |   by host   |      |
+-----+   |       |             |      |   bridge    |      |   +--------+
|     |   |       |             +------+             |      |   |        |
| CPU |   |       |             | RAM  |             |      |   | Device |
|     |   |       |             |      |             |      |   |        |
+-----+   +-------+             +------+             +------+   +--------+
          |       |  Virtual    |Buffer|   Mapping   |      |
        X +-------+ --------> Y +------+ <---------- +------+ Z
          |       |  mapping    | RAM  |   by IOMMU
          |       |             |      |
          |       |             |      |
          +-------+             +------+
```

During the enumeration process, the kernel learns about I/O devices and their MMIO space and the host bridges that connect them to the system. For example, if a PCI device has a BAR, the kernel reads the bus address (A) from the BAR and converts it to a CPU physical address (B). The address B is stored in a struct resource and usually exposed via /proc/iomem. When a driver claims a device, it typically uses ioremap() to map physical address B at a virtual address (C). It can then use, e.g., ioread32(C), to access the device registers at bus address A.

If the device supports DMA, the driver sets up a buffer using kmalloc() or a similar interface, which returns a virtual address (X). The virtual memory system maps X to a physical address (Y) in system RAM. The driver can use virtual address X to access the buffer, but the device itself cannot because DMA doesn't go through the CPU virtual memory system.

In some simple systems, the device can do DMA directly to physical address Y. But in many others, there is IOMMU hardware that translates DMA addresses to physical addresses, e.g., it translates Z to Y. This is part of the reason for the DMA API: the driver can give a virtual address X to an interface like dma_map_single(), which sets up any required IOMMU mapping and returns the DMA address Z. The driver then tells the device to do DMA to Z, and the IOMMU maps it to the buffer at address Y in system RAM.

So that Linux can use the dynamic DMA mapping, it needs some help from the drivers, namely it has to take into account that DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer.

The following API will work of course even on platforms where no such hardware exists.

Note that the DMA API works with any bus independent of the underlying microprocessor architecture. You should use the DMA API rather than the bus-specific DMA API, i.e., use the dma_map_*() interfaces rather than the pci_map_*() interfaces.

First of all, you should make sure:

```
#include <linux/dma-mapping.h>
```

is in your driver, which provides the definition of dma_addr_t. This type can hold any valid DMA address for the platform and should be used everywhere you hold a DMA address returned from the DMA mapping functions.

6.4.2 What memory is DMA'able?

The first piece of information you must know is what kernel memory can be used with the DMA mapping facilities. There has been an unwritten set of rules regarding this, and this text is an attempt to finally write them down.

If you acquired your memory via the page allocator (i.e. __get_free_page*()) or the generic memory allocators (i.e. kmalloc() or kmem_cache_alloc()) then you may DMA to/from that memory using the addresses returned from those routines.

This means specifically that you may _not_ use the memory/addresses returned from vmalloc() for DMA. It is possible to DMA to the _underlying_ memory mapped into a vmalloc() area, but this requires walking page tables to get the physical addresses, and then translating each of those pages back to a kernel address using something like __va(). [EDIT: Update this when we integrate Gerd Knorr's generic code which does this.]

This rule also means that you may use neither kernel image addresses (items in data/text/bss segments), nor module image addresses, nor stack addresses for DMA. These could all be mapped somewhere entirely different than the rest of physical memory. Even if those classes of memory could physically work with DMA, you'd need to ensure the I/O buffers were cacheline-aligned. Without that, you'd see cacheline sharing problems (data corruption) on CPUs with DMA-incoherent caches. (The CPU could write to one word, DMA would write to a different one in the same cache line, and one of them could be overwritten.)

Also, this means that you cannot take the return of a kmap() call and DMA to/from that. This is similar to vmalloc().

What about block I/O and networking buffers? The block I/O and networking subsystems make sure that the buffers they use are valid for you to DMA from/to.

6.4.3 DMA addressing capabilities

By default, the kernel assumes that your device can address 32-bits of DMA addressing. For a 64-bit capable device, this needs to be increased, and for a device with limitations, it needs to be decreased.

Special note about PCI: PCI-X specification requires PCI-X devices to support 64-bit addressing (DAC) for all transactions. And at least one platform (SGI SN2) requires 64-bit consistent allocations to operate correctly when the IO bus is in PCI-X mode.

For correct operation, you must set the DMA mask to inform the kernel about your devices DMA addressing capabilities.

This is performed via a call to dma_set_mask_and_coherent():

```
int dma_set_mask_and_coherent(struct device *dev, u64 mask);
```

which will set the mask for both streaming and coherent APIs together. If you have some special requirements, then the following two separate calls can be used instead:

The setup for streaming mappings is performed via a call to dma_set_mask():

```
int dma_set_mask(struct device *dev, u64 mask);
```

The setup for consistent allocations is performed via a call to dma_set_coherent_mask():

```
int dma_set_coherent_mask(struct device *dev, u64 mask);
```

Here, dev is a pointer to the device struct of your device, and mask is a bit mask describing which bits of an address your device supports. Often the device struct of your device is embedded in the bus-specific device struct of your device. For example, &pdev->dev is a pointer to the device struct of a PCI device (pdev is a pointer to the PCI device struct of your device).

These calls usually return zero to indicate your device can perform DMA properly on the machine given the address mask you provided, but they might return an error if the mask is too small to be supportable on the given system. If it returns non-zero, your device cannot perform DMA properly on this platform, and attempting to do so will result in undefined behavior. You must not use DMA on this device unless the dma_set_mask family of functions has returned success.

This means that in the failure case, you have two options:

1) Use some non-DMA mode for data transfer, if possible.
2) Ignore this device and do not initialize it.

It is recommended that your driver print a kernel KERN_WARNING message when setting the DMA mask fails. In this manner, if a user of your driver reports that performance is bad or that the device is not even detected, you can ask them for the kernel messages to find out exactly why.

The standard 64-bit addressing device would do something like this:

```
if (dma_set_mask_and_coherent(dev, DMA_BIT_MASK(64))) {
        dev_warn(dev, "mydev: No suitable DMA available\n");
        goto ignore_this_device;
}
```

If the device only supports 32-bit addressing for descriptors in the coherent allocations, but supports full 64-bits for streaming mappings it would look like this:

```
if (dma_set_mask(dev, DMA_BIT_MASK(64))) {
        dev_warn(dev, "mydev: No suitable DMA available\n");
        goto ignore_this_device;
}
```

The coherent mask will always be able to set the same or a smaller mask as the streaming mask. However for the rare case that a device driver only uses consistent allocations, one would have to check the return value from dma_set_coherent_mask().

Finally, if your device can only drive the low 24-bits of address you might do something like:

```
if (dma_set_mask(dev, DMA_BIT_MASK(24))) {
        dev_warn(dev, "mydev: 24-bit DMA addressing not available\n");
        goto ignore_this_device;
}
```

When dma_set_mask() or dma_set_mask_and_coherent() is successful, and returns zero, the kernel saves away this mask you have provided. The kernel will use this information later when you make DMA mappings.

There is a case which we are aware of at this time, which is worth mentioning in this documentation. If your device supports multiple functions (for example a sound card provides playback and record functions) and the various different functions have _different_ DMA addressing limitations, you may wish to probe each mask and only provide the functionality which the machine can handle. It is important that the last call to dma_set_mask() be for the most specific mask.

Here is pseudo-code showing how this might be done:

```
#define PLAYBACK_ADDRESS_BITS   DMA_BIT_MASK(32)
#define RECORD_ADDRESS_BITS     DMA_BIT_MASK(24)

struct my_sound_card *card;
struct device *dev;

...
if (!dma_set_mask(dev, PLAYBACK_ADDRESS_BITS)) {
        card->playback_enabled = 1;
} else {
        card->playback_enabled = 0;
        dev_warn(dev, "%s: Playback disabled due to DMA limitations\n",
               card->name);
}
if (!dma_set_mask(dev, RECORD_ADDRESS_BITS)) {
        card->record_enabled = 1;
} else {
        card->record_enabled = 0;
        dev_warn(dev, "%s: Record disabled due to DMA limitations\n",
               card->name);
}
```

A sound card was used as an example here because this genre of PCI devices seems to be littered with ISA chips given a PCI front end, and thus retaining the 16MB DMA addressing limitations of ISA.

6.4.4 Types of DMA mappings

There are two types of DMA mappings:

- Consistent DMA mappings which are usually mapped at driver initialization, unmapped at the end and for which the hardware should guarantee that the device and the CPU can access the data in parallel and will see updates made by each other without any explicit software flushing.

  Think of "consistent" as "synchronous" or "coherent".

  The current default is to return consistent memory in the low 32 bits of the DMA space. However, for future compatibility you should set the consistent mask even if this default is fine for your driver.

  Good examples of what to use consistent mappings for are:

  - Network card DMA ring descriptors.
  - SCSI adapter mailbox command data structures.
  - Device firmware microcode executed out of main memory.

  The invariant these examples all require is that any CPU store to memory is immediately visible to the device, and vice versa. Consistent mappings guarantee this.

  Important: Consistent DMA memory does not preclude the usage of proper memory barriers. The CPU may reorder stores to consistent memory just as it may normal memory. Example: if it is important for the device to see the first word of a descriptor updated before the second, you must do something like:

  ```
  desc->word0 = address;
  wmb();
  desc->word1 = DESC_VALID;
  ```

  in order to get correct behavior on all platforms.

  Also, on some platforms your driver may need to flush CPU write buffers in much the same way as it needs to flush write buffers found in PCI bridges (such as by reading a register's value after writing it).

- Streaming DMA mappings which are usually mapped for one DMA transfer, unmapped right after it (unless you use dma_sync_* below) and for which hardware can optimize for sequential accesses.

  Think of "streaming" as "asynchronous" or "outside the coherency domain".

  Good examples of what to use streaming mappings for are:

  - Networking buffers transmitted/received by a device.
  - Filesystem buffers written/read by a SCSI device.

  The interfaces for using this type of mapping were designed in such a way that an implementation can make whatever performance optimizations the hardware allows. To this end, when using such mappings you must be explicit about what you want to happen.

Neither type of DMA mapping has alignment restrictions that come from the underlying bus, although some devices may have such restrictions. Also, systems with caches that aren't DMA-coherent will work better when the underlying buffers don't share cache lines with other data.

6.4.5 Using Consistent DMA mappings

To allocate and map large (PAGE_SIZE or so) consistent DMA regions, you should do:

```
dma_addr_t dma_handle;

cpu_addr = dma_alloc_coherent(dev, size, &dma_handle, gfp);
```

where device is a `struct device *`. This may be called in interrupt context with the GFP_ATOMIC flag.

Size is the length of the region you want to allocate, in bytes.

This routine will allocate RAM for that region, so it acts similarly to __get_free_pages() (but takes size instead of a page order). If your driver needs regions sized smaller than a page, you may prefer using the dma_pool interface, described below.

The consistent DMA mapping interfaces, will by default return a DMA address which is 32-bit addressable. Even if the device indicates (via the DMA mask) that it may address the upper 32-bits, consistent allocation will only return > 32-bit addresses for DMA if the consistent DMA mask has been explicitly changed via dma_set_coherent_mask(). This is true of the dma_pool interface as well.

dma_alloc_coherent() returns two values: the virtual address which you can use to access it from the CPU and dma_handle which you pass to the card.

The CPU virtual address and the DMA address are both guaranteed to be aligned to the smallest PAGE_SIZE order which is greater than or equal to the requested size. This invariant exists (for example) to guarantee that if you allocate a chunk which is smaller than or equal to 64 kilobytes, the extent of the buffer you receive will not cross a 64K boundary.

To unmap and free such a DMA region, you call:

```
dma_free_coherent(dev, size, cpu_addr, dma_handle);
```

where dev, size are the same as in the above call and cpu_addr and dma_handle are the values dma_alloc_coherent() returned to you. This function may not be called in interrupt context.

If your driver needs lots of smaller memory regions, you can write custom code to subdivide pages returned by dma_alloc_coherent(), or you can use the dma_pool API to do that. A dma_pool is like a kmem_cache, but it uses dma_alloc_coherent(), not __get_free_pages(). Also, it understands common hardware constraints for alignment, like queue heads needing to be aligned on N byte boundaries.

Create a dma_pool like this:

```
struct dma_pool *pool;

pool = dma_pool_create(name, dev, size, align, boundary);
```

The "name" is for diagnostics (like a kmem_cache name); dev and size are as above. The device's hardware alignment requirement for this type of data is "align" (which is expressed in bytes, and must be a power of two). If your device has no boundary crossing restrictions, pass 0 for boundary; passing 4096 says memory allocated from this pool must not cross 4KByte boundaries (but at that time it may be better to use dma_alloc_coherent() directly instead).

Allocate memory from a DMA pool like this:

```
cpu_addr = dma_pool_alloc(pool, flags, &dma_handle);
```

flags are GFP_KERNEL if blocking is permitted (not in_interrupt nor holding SMP locks), GFP_ATOMIC otherwise. Like dma_alloc_coherent(), this returns two values, cpu_addr and dma_handle.

Free memory that was allocated from a dma_pool like this:

```
dma_pool_free(pool, cpu_addr, dma_handle);
```

where pool is what you passed to *dma_pool_alloc()*, and cpu_addr and dma_handle are the values *dma_pool_alloc()* returned. This function may be called in interrupt context.

Destroy a dma_pool by calling:

```
dma_pool_destroy(pool);
```

Make sure you've called *dma_pool_free()* for all memory allocated from a pool before you destroy the pool. This function may not be called in interrupt context.

6.4.6 DMA Direction

The interfaces described in subsequent portions of this document take a DMA direction argument, which is an integer and takes on one of the following values:

```
DMA_BIDIRECTIONAL
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_NONE
```

You should provide the exact DMA direction if you know it.

DMA_TO_DEVICE means "from main memory to the device" DMA_FROM_DEVICE means "from the device to main memory" It is the direction in which the data moves during the DMA transfer.

You are _strongly_ encouraged to specify this as precisely as you possibly can.

If you absolutely cannot know the direction of the DMA transfer, specify DMA_BIDIRECTIONAL. It means that the DMA can go in either direction. The platform guarantees that you may legally specify this, and that it will work, but this may be at the cost of performance for example.

The value DMA_NONE is to be used for debugging. One can hold this in a data structure before you come to know the precise direction, and this will help catch cases where your direction tracking logic has failed to set things up properly.

Another advantage of specifying this value precisely (outside of potential platform-specific optimizations of such) is for debugging. Some platforms actually have a write permission boolean which DMA mappings can be marked with, much like page protections in the user program address space. Such platforms can and do report errors in the kernel logs when the DMA controller hardware detects violation of the permission setting.

Only streaming mappings specify a direction, consistent mappings implicitly have a direction attribute setting of DMA_BIDIRECTIONAL.

The SCSI subsystem tells you the direction to use in the 'sc_data_direction' member of the SCSI command your driver is working on.

For Networking drivers, it's a rather simple affair. For transmit packets, map/unmap them with the DMA_TO_DEVICE direction specifier. For receive packets, just the opposite, map/unmap them with the DMA_FROM_DEVICE direction specifier.

6.4.7 Using Streaming DMA mappings

The streaming DMA mapping routines can be called from interrupt context. There are two versions of each map/unmap, one which will map/unmap a single memory region, and one which will map/unmap a scatterlist.

To map a single region, you do:

```
struct device *dev = &my_dev->dev;
dma_addr_t dma_handle;
void *addr = buffer->ptr;
size_t size = buffer->len;

dma_handle = dma_map_single(dev, addr, size, direction);
if (dma_mapping_error(dev, dma_handle)) {
        /*
         * reduce current DMA mapping usage,
         * delay and try again later or
         * reset driver.
         */
        goto map_error_handling;
}
```

and to unmap it:

```
dma_unmap_single(dev, dma_handle, size, direction);
```

You should call dma_mapping_error() as dma_map_single() could fail and return error. Doing so will ensure that the mapping code will work correctly on all DMA implementations without any dependency on the specifics of the underlying implementation. Using the returned address without checking for errors could result in failures ranging from panics to silent data corruption. The same applies to dma_map_page() as well.

You should call dma_unmap_single() when the DMA activity is finished, e.g., from the interrupt which told you that the DMA transfer is done.

Using CPU pointers like this for single mappings has a disadvantage: you cannot reference HIGHMEM memory in this way. Thus, there is a map/unmap interface pair akin to dma_{map,unmap}_single(). These interfaces deal with page/offset pairs instead of CPU pointers. Specifically:

```
struct device *dev = &my_dev->dev;
dma_addr_t dma_handle;
struct page *page = buffer->page;
unsigned long offset = buffer->offset;
size_t size = buffer->len;

dma_handle = dma_map_page(dev, page, offset, size, direction);
if (dma_mapping_error(dev, dma_handle)) {
        /*
         * reduce current DMA mapping usage,
         * delay and try again later or
         * reset driver.
         */
        goto map_error_handling;
}

...

dma_unmap_page(dev, dma_handle, size, direction);
```

Here, "offset" means byte offset within the given page.

You should call dma_mapping_error() as dma_map_page() could fail and return error as outlined under the dma_map_single() discussion.

You should call dma_unmap_page() when the DMA activity is finished, e.g., from the interrupt which told you that the DMA transfer is done.

With scatterlists, you map a region gathered from several regions by:

```
int i, count = dma_map_sg(dev, sglist, nents, direction);
struct scatterlist *sg;

for_each_sg(sglist, sg, count, i) {
        hw_address[i] = sg_dma_address(sg);
        hw_len[i] = sg_dma_len(sg);
}
```

where nents is the number of entries in the sglist.

The implementation is free to merge several consecutive sglist entries into one (e.g. if DMA mapping is done with PAGE_SIZE granularity, any consecutive sglist entries can be merged into one provided the first one ends and the second one starts on a page boundary - in fact this is a huge advantage for cards which either cannot do scatter-gather or have very limited number of scatter-gather entries) and returns the actual number of sg entries it mapped them to. On failure 0 is returned.

Then you should loop count times (note: this can be less than nents times) and use sg_dma_address() and sg_dma_len() macros where you previously accessed sg->address and sg->length as shown above.

To unmap a scatterlist, just call:

```
dma_unmap_sg(dev, sglist, nents, direction);
```

Again, make sure DMA activity has already finished.

Note: The 'nents' argument to the dma_unmap_sg call must be the _same_ one you passed into the dma_map_sg call, it should _NOT_ be the 'count' value _returned_ from the dma_map_sg call.

Every dma_map_{single,sg}() call should have its dma_unmap_{single,sg}() counterpart, because the DMA address space is a shared resource and you could render the machine unusable by consuming all DMA addresses.

If you need to use the same streaming DMA region multiple times and touch the data in between the DMA transfers, the buffer needs to be synced properly in order for the CPU and device to see the most up-to-date and correct copy of the DMA buffer.

So, firstly, just map it with dma_map_{single,sg}(), and after each DMA transfer call either:

```
dma_sync_single_for_cpu(dev, dma_handle, size, direction);
```

or:

```
dma_sync_sg_for_cpu(dev, sglist, nents, direction);
```

as appropriate.

Then, if you wish to let the device get at the DMA area again, finish accessing the data with the CPU, and then before actually giving the buffer to the hardware call either:

```
dma_sync_single_for_device(dev, dma_handle, size, direction);
```

or:

```
dma_sync_sg_for_device(dev, sglist, nents, direction);
```

as appropriate.

Note: The 'nents' argument to dma_sync_sg_for_cpu() and dma_sync_sg_for_device() must be the same passed to dma_map_sg(). It is _NOT_ the count returned by dma_map_sg().

After the last DMA transfer call one of the DMA unmap routines dma_unmap_{single,sg}(). If you don't touch the data from the first dma_map_*() call till dma_unmap_*(), then you don't have to call the dma_sync_*() routines at all.

Here is pseudo code which shows a situation in which you would need to use the dma_sync_*() interfaces:

```
my_card_setup_receive_buffer(struct my_card *cp, char *buffer, int len)
{
        dma_addr_t mapping;

        mapping = dma_map_single(cp->dev, buffer, len, DMA_FROM_DEVICE);
```

```
        if (dma_mapping_error(cp->dev, mapping)) {
                /*
                 * reduce current DMA mapping usage,
                 * delay and try again later or
                 * reset driver.
                 */
                goto map_error_handling;
        }

        cp->rx_buf = buffer;
        cp->rx_len = len;
        cp->rx_dma = mapping;

        give_rx_buf_to_card(cp);
}

...

my_card_interrupt_handler(int irq, void *devid, struct pt_regs *regs)
{
        struct my_card *cp = devid;

        ...
        if (read_card_status(cp) == RX_BUF_TRANSFERRED) {
                struct my_card_header *hp;

                /* Examine the header to see if we wish
                 * to accept the data.  But synchronize
                 * the DMA transfer with the CPU first
                 * so that we see updated contents.
                 */
                dma_sync_single_for_cpu(&cp->dev, cp->rx_dma,
                                        cp->rx_len,
                                        DMA_FROM_DEVICE);

                /* Now it is safe to examine the buffer. */
                hp = (struct my_card_header *) cp->rx_buf;
                if (header_is_ok(hp)) {
                        dma_unmap_single(&cp->dev, cp->rx_dma, cp->rx_len,
                                         DMA_FROM_DEVICE);
                        pass_to_upper_layers(cp->rx_buf);
                        make_and_setup_new_rx_buf(cp);
                } else {
                        /* CPU should not write to
                         * DMA_FROM_DEVICE-mapped area,
                         * so dma_sync_single_for_device() is
                         * not needed here. It would be required
                         * for DMA_BIDIRECTIONAL mapping if
                         * the memory was modified.
                         */
```

```
                        give_rx_buf_to_card(cp);
                }
        }
}
```

6.4.8 Handling Errors

DMA address space is limited on some architectures and an allocation failure can be determined by:

- checking if dma_alloc_coherent() returns NULL or dma_map_sg returns 0
- checking the dma_addr_t returned from dma_map_single() and dma_map_page() by using dma_mapping_error():

  ```
  dma_addr_t dma_handle;

  dma_handle = dma_map_single(dev, addr, size, direction);
  if (dma_mapping_error(dev, dma_handle)) {
          /*
           * reduce current DMA mapping usage,
           * delay and try again later or
           * reset driver.
           */
          goto map_error_handling;
  }
  ```

- unmap pages that are already mapped, when mapping error occurs in the middle of a multiple page mapping attempt. These example are applicable to dma_map_page() as well.

Example 1:

```
dma_addr_t dma_handle1;
dma_addr_t dma_handle2;

dma_handle1 = dma_map_single(dev, addr, size, direction);
if (dma_mapping_error(dev, dma_handle1)) {
        /*
         * reduce current DMA mapping usage,
         * delay and try again later or
         * reset driver.
         */
        goto map_error_handling1;
}
dma_handle2 = dma_map_single(dev, addr, size, direction);
if (dma_mapping_error(dev, dma_handle2)) {
        /*
         * reduce current DMA mapping usage,
         * delay and try again later or
         * reset driver.
         */
```

```
        goto map_error_handling2;
}

...

map_error_handling2:
        dma_unmap_single(dma_handle1);
map_error_handling1:
```

Example 2:

```
/*
 * if buffers are allocated in a loop, unmap all mapped buffers when
 * mapping error is detected in the middle
 */

dma_addr_t dma_addr;
dma_addr_t array[DMA_BUFFERS];
int save_index = 0;

for (i = 0; i < DMA_BUFFERS; i++) {

        ...

        dma_addr = dma_map_single(dev, addr, size, direction);
        if (dma_mapping_error(dev, dma_addr)) {
                /*
                 * reduce current DMA mapping usage,
                 * delay and try again later or
                 * reset driver.
                 */
                goto map_error_handling;
        }
        array[i].dma_addr = dma_addr;
        save_index++;
}

...

map_error_handling:

for (i = 0; i < save_index; i++) {

        ...

        dma_unmap_single(array[i].dma_addr);
}
```

Networking drivers must call dev_kfree_skb() to free the socket buffer and return NETDEV_TX_OK if the DMA mapping fails on the transmit hook (ndo_start_xmit). This means that the socket buffer is just dropped in the failure case.

SCSI drivers must return SCSI_MLQUEUE_HOST_BUSY if the DMA mapping fails in the queuecommand hook. This means that the SCSI subsystem passes the command to the driver again later.

6.4.9 Optimizing Unmap State Space Consumption

On many platforms, dma_unmap_{single,page}() is simply a nop. Therefore, keeping track of the mapping address and length is a waste of space. Instead of filling your drivers up with ifdefs and the like to "work around" this (which would defeat the whole purpose of a portable API) the following facilities are provided.

Actually, instead of describing the macros one by one, we'll transform some example code.

1) Use DEFINE_DMA_UNMAP_{ADDR,LEN} in state saving structures. Example, before:

```
struct ring_state {
        struct sk_buff *skb;
        dma_addr_t mapping;
        __u32 len;
};
```

after:

```
struct ring_state {
        struct sk_buff *skb;
        DEFINE_DMA_UNMAP_ADDR(mapping);
        DEFINE_DMA_UNMAP_LEN(len);
};
```

2) Use dma_unmap_{addr,len}_set() to set these values. Example, before:

```
ringp->mapping = FOO;
ringp->len = BAR;
```

after:

```
dma_unmap_addr_set(ringp, mapping, FOO);
dma_unmap_len_set(ringp, len, BAR);
```

3) Use dma_unmap_{addr,len}() to access these values. Example, before:

```
dma_unmap_single(dev, ringp->mapping, ringp->len,
                 DMA_FROM_DEVICE);
```

after:

```
dma_unmap_single(dev,
                 dma_unmap_addr(ringp, mapping),
                 dma_unmap_len(ringp, len),
                 DMA_FROM_DEVICE);
```

It really should be self-explanatory. We treat the ADDR and LEN separately, because it is possible for an implementation to only need the address in order to perform the unmap operation.

6.4.10 Platform Issues

If you are just writing drivers for Linux and do not maintain an architecture port for the kernel, you can safely skip down to "Closing".

1) Struct scatterlist requirements.

   You need to enable CONFIG_NEED_SG_DMA_LENGTH if the architecture supports IOMMUs (including software IOMMU).

2) ARCH_DMA_MINALIGN

   Architectures must ensure that kmalloc'ed buffer is DMA-safe. Drivers and subsystems depend on it. If an architecture isn't fully DMA-coherent (i.e. hardware doesn't ensure that data in the CPU cache is identical to data in main memory), ARCH_DMA_MINALIGN must be set so that the memory allocator makes sure that kmalloc'ed buffer doesn't share a cache line with the others. See arch/arm/include/asm/cache.h as an example.

   Note that ARCH_DMA_MINALIGN is about DMA memory alignment constraints. You don't need to worry about the architecture data alignment constraints (e.g. the alignment constraints about 64-bit objects).

6.4.11 Closing

This document, and the API itself, would not be in its current form without the feedback and suggestions from numerous individuals. We would like to specifically mention, in no particular order, the following people:

```
Russell King <rmk@arm.linux.org.uk>
Leo Dagum <dagum@barrel.engr.sgi.com>
Ralf Baechle <ralf@oss.sgi.com>
Grant Grundler <grundler@cup.hp.com>
Jay Estabrook <Jay.Estabrook@compaq.com>
Thomas Sailer <sailer@ife.ee.ethz.ch>
Andrea Arcangeli <andrea@suse.de>
Jens Axboe <jens.axboe@oracle.com>
David Mosberger-Tang <davidm@hpl.hp.com>
```

6.5 DMA attributes

This document describes the semantics of the DMA attributes that are defined in linux/dma-mapping.h.

6.5.1 DMA_ATTR_WEAK_ORDERING

DMA_ATTR_WEAK_ORDERING specifies that reads and writes to the mapping may be weakly ordered, that is that reads and writes may pass each other.

Since it is optional for platforms to implement DMA_ATTR_WEAK_ORDERING, those that do not will simply ignore the attribute and exhibit default behavior.

6.5.2 DMA_ATTR_WRITE_COMBINE

DMA_ATTR_WRITE_COMBINE specifies that writes to the mapping may be buffered to improve performance.

Since it is optional for platforms to implement DMA_ATTR_WRITE_COMBINE, those that do not will simply ignore the attribute and exhibit default behavior.

6.5.3 DMA_ATTR_NO_KERNEL_MAPPING

DMA_ATTR_NO_KERNEL_MAPPING lets the platform to avoid creating a kernel virtual mapping for the allocated buffer. On some architectures creating such mapping is non-trivial task and consumes very limited resources (like kernel virtual address space or dma consistent address space). Buffers allocated with this attribute can be only passed to user space by calling dma_mmap_attrs(). By using this API, you are guaranteeing that you won't dereference the pointer returned by dma_alloc_attr(). You can treat it as a cookie that must be passed to dma_mmap_attrs() and dma_free_attrs(). Make sure that both of these also get this attribute set on each call.

Since it is optional for platforms to implement DMA_ATTR_NO_KERNEL_MAPPING, those that do not will simply ignore the attribute and exhibit default behavior.

6.5.4 DMA_ATTR_SKIP_CPU_SYNC

By default dma_map_{single,page,sg} functions family transfer a given buffer from CPU domain to device domain. Some advanced use cases might require sharing a buffer between more than one device. This requires having a mapping created separately for each device and is usually performed by calling dma_map_{single,page,sg} function more than once for the given buffer with device pointer to each device taking part in the buffer sharing. The first call transfers a buffer from 'CPU' domain to 'device' domain, what synchronizes CPU caches for the given region (usually it means that the cache has been flushed or invalidated depending on the dma direction). However, next calls to dma_map_{single,page,sg}() for other devices will perform exactly the same synchronization operation on the CPU cache. CPU cache synchronization might be a time consuming operation, especially if the buffers are large, so it is highly recommended to avoid it if possible. DMA_ATTR_SKIP_CPU_SYNC allows platform code to skip synchronization of the CPU cache for the given buffer assuming that it has been already transferred to 'device' domain. This attribute can be also used for dma_unmap_{single,page,sg} functions family to force buffer to stay in device domain after releasing a mapping for it. Use this attribute with care!

6.5.5 DMA_ATTR_FORCE_CONTIGUOUS

By default DMA-mapping subsystem is allowed to assemble the buffer allocated by dma_alloc_attrs() function from individual pages if it can be mapped as contiguous chunk into device dma address space. By specifying this attribute the allocated buffer is forced to be contiguous also in physical memory.

6.5.6 DMA_ATTR_ALLOC_SINGLE_PAGES

This is a hint to the DMA-mapping subsystem that it's probably not worth the time to try to allocate memory to in a way that gives better TLB efficiency (AKA it's not worth trying to build the mapping out of larger pages). You might want to specify this if:

- You know that the accesses to this memory won't thrash the TLB. You might know that the accesses are likely to be sequential or that they aren't sequential but it's unlikely you'll ping-pong between many addresses that are likely to be in different physical pages.
- You know that the penalty of TLB misses while accessing the memory will be small enough to be inconsequential. If you are doing a heavy operation like decryption or decompression this might be the case.
- You know that the DMA mapping is fairly transitory. If you expect the mapping to have a short lifetime then it may be worth it to optimize allocation (avoid coming up with large pages) instead of getting the slight performance win of larger pages.

Setting this hint doesn't guarantee that you won't get huge pages, but it means that we won't try quite as hard to get them.

Note: At the moment DMA_ATTR_ALLOC_SINGLE_PAGES is only implemented on ARM, though ARM64 patches will likely be posted soon.

6.5.7 DMA_ATTR_NO_WARN

This tells the DMA-mapping subsystem to suppress allocation failure reports (similarly to __GFP_NOWARN).

On some architectures allocation failures are reported with error messages to the system logs. Although this can help to identify and debug problems, drivers which handle failures (eg, retry later) have no problems with them, and can actually flood the system logs with error messages that aren't any problem at all, depending on the implementation of the retry mechanism.

So, this provides a way for drivers to avoid those error messages on calls where allocation failures are not a problem, and shouldn't bother the logs.

Note: At the moment DMA_ATTR_NO_WARN is only implemented on PowerPC.

6.5.8 DMA_ATTR_PRIVILEGED

Some advanced peripherals such as remote processors and GPUs perform accesses to DMA buffers in both privileged "supervisor" and unprivileged "user" modes. This attribute is used to indicate to the DMA-mapping subsystem that the buffer is fully accessible at the elevated privilege level (and ideally inaccessible or at least read-only at the lesser-privileged levels).

6.6 DMA with ISA and LPC devices

Author
Pierre Ossman <drzeus@drzeus.cx>

This document describes how to do DMA transfers using the old ISA DMA controller. Even though ISA is more or less dead today the LPC bus uses the same DMA system so it will be around for quite some time.

6.6.1 Headers and dependencies

To do ISA style DMA you need to include two headers:

```
#include <linux/dma-mapping.h>
#include <asm/dma.h>
```

The first is the generic DMA API used to convert virtual addresses to bus addresses (see *Dynamic DMA mapping using the generic device* for details).

The second contains the routines specific to ISA DMA transfers. Since this is not present on all platforms make sure you construct your Kconfig to be dependent on ISA_DMA_API (not ISA) so that nobody tries to build your driver on unsupported platforms.

6.6.2 Buffer allocation

The ISA DMA controller has some very strict requirements on which memory it can access so extra care must be taken when allocating buffers.

(You usually need a special buffer for DMA transfers instead of transferring directly to and from your normal data structures.)

The DMA-able address space is the lowest 16 MB of _physical_ memory. Also the transfer block may not cross page boundaries (which are 64 or 128 KiB depending on which channel you use).

In order to allocate a piece of memory that satisfies all these requirements you pass the flag GFP_DMA to kmalloc.

Unfortunately the memory available for ISA DMA is scarce so unless you allocate the memory during boot-up it's a good idea to also pass __GFP_RETRY_MAYFAIL and __GFP_NOWARN to make the allocator try a bit harder.

(This scarcity also means that you should allocate the buffer as early as possible and not release it until the driver is unloaded.)

6.6.3 Address translation

To translate the virtual address to a bus address, use the normal DMA API. Do _not_ use isa_virt_to_bus() even though it does the same thing. The reason for this is that the function isa_virt_to_bus() will require a Kconfig dependency to ISA, not just ISA_DMA_API which is really all you need. Remember that even though the DMA controller has its origins in ISA it is used elsewhere.

Note: x86_64 had a broken DMA API when it came to ISA but has since been fixed. If your arch has problems then fix the DMA API instead of reverting to the ISA functions.

6.6.4 Channels

A normal ISA DMA controller has 8 channels. The lower four are for 8-bit transfers and the upper four are for 16-bit transfers.

(Actually the DMA controller is really two separate controllers where channel 4 is used to give DMA access for the second controller (0-3). This means that of the four 16-bits channels only three are usable.)

You allocate these in a similar fashion as all basic resources:

extern int request_dma(unsigned int dmanr, const char * device_id); extern void free_dma(unsigned int dmanr);

The ability to use 16-bit or 8-bit transfers is _not_ up to you as a driver author but depends on what the hardware supports. Check your specs or test different channels.

6.6.5 Transfer data

Now for the good stuff, the actual DMA transfer. :)

Before you use any ISA DMA routines you need to claim the DMA lock using claim_dma_lock(). The reason is that some DMA operations are not atomic so only one driver may fiddle with the registers at a time.

The first time you use the DMA controller you should call clear_dma_ff(). This clears an internal register in the DMA controller that is used for the non-atomic operations. As long as you (and everyone else) uses the locking functions then you only need to reset this once.

Next, you tell the controller in which direction you intend to do the transfer using set_dma_mode(). Currently you have the options DMA_MODE_READ and DMA_MODE_WRITE.

Set the address from where the transfer should start (this needs to be 16-bit aligned for 16-bit transfers) and how many bytes to transfer. Note that it's _bytes_. The DMA routines will do all the required translation to values that the DMA controller understands.

The final step is enabling the DMA channel and releasing the DMA lock.

Once the DMA transfer is finished (or timed out) you should disable the channel again. You should also check get_dma_residue() to make sure that all data has been transferred.

Example:

```
int flags, residue;

flags = claim_dma_lock();

clear_dma_ff();

set_dma_mode(channel, DMA_MODE_WRITE);
set_dma_addr(channel, phys_addr);
set_dma_count(channel, num_bytes);

dma_enable(channel);

release_dma_lock(flags);

while (!device_done());

flags = claim_dma_lock();

dma_disable(channel);

residue = dma_get_residue(channel);
if (residue != 0)
        printk(KERN_ERR "driver: Incomplete DMA transfer!"
                " %d bytes left!\n", residue);

release_dma_lock(flags);
```

6.6.6 Suspend/resume

It is the driver's responsibility to make sure that the machine isn't suspended while a DMA transfer is in progress. Also, all DMA settings are lost when the system suspends so if your driver relies on the DMA controller being in a certain state then you have to restore these registers upon resume.

6.7 Memory Management APIs

6.7.1 User Space Memory Access

get_user

`get_user (x, ptr)`

Get a simple variable from user space.

Parameters

x

Variable to store result.

ptr
Source address, in user space.

Context

User context only. This function may sleep if pagefaults are enabled.

Description

This macro copies a single simple variable from user space to kernel space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and the result of dereferencing **ptr** must be assignable to **x** without a cast.

Return

zero on success, or -EFAULT on error. On error, the variable **x** is set to zero.

__get_user

`__get_user (x, ptr)`

Get a simple variable from user space, with less checking.

Parameters

x
Variable to store result.

ptr
Source address, in user space.

Context

User context only. This function may sleep if pagefaults are enabled.

Description

This macro copies a single simple variable from user space to kernel space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and the result of dereferencing **ptr** must be assignable to **x** without a cast.

Caller must check the pointer with access_ok() before calling this function.

Return

zero on success, or -EFAULT on error. On error, the variable **x** is set to zero.

put_user

`put_user (x, ptr)`

Write a simple value into user space.

Parameters

x
Value to copy to user space.

ptr
Destination address, in user space.

Context

User context only. This function may sleep if pagefaults are enabled.

Description

This macro copies a single simple value from kernel space to user space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and **x** must be assignable to the result of dereferencing **ptr**.

Return

zero on success, or -EFAULT on error.

`__put_user`

`__put_user (x, ptr)`

Write a simple value into user space, with less checking.

Parameters

`x`
Value to copy to user space.

`ptr`
Destination address, in user space.

Context

User context only. This function may sleep if pagefaults are enabled.

Description

This macro copies a single simple value from kernel space to user space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and **x** must be assignable to the result of dereferencing **ptr**.

Caller must check the pointer with access_ok() before calling this function.

Return

zero on success, or -EFAULT on error.

unsigned long **`clear_user`**(void __user *to, unsigned long n)

Zero a block of memory in user space.

Parameters

`void __user *to`
Destination address, in user space.

`unsigned long n`
Number of bytes to zero.

Description

Zero a block of memory in user space.

Return

number of bytes that could not be cleared. On success, this will be zero.

unsigned long **__clear_user**(void __user *to, unsigned long n)

Zero a block of memory in user space, with less checking.

Parameters

void __user *to
Destination address, in user space.

unsigned long n
Number of bytes to zero.

Description

Zero a block of memory in user space. Caller must check the specified block with access_ok() before calling this function.

Return

number of bytes that could not be cleared. On success, this will be zero.

int **get_user_pages_fast**(unsigned long start, int nr_pages, unsigned int gup_flags, struct page **pages)

pin user pages in memory

Parameters

unsigned long start
starting user address

int nr_pages
number of pages from start to pin

unsigned int gup_flags
flags modifying pin behaviour

struct page **pages
array that receives pointers to the pages pinned. Should be at least nr_pages long.

Description

Attempt to pin user pages in memory without taking mm->mmap_lock. If not successful, it will fall back to taking the lock and calling get_user_pages().

Returns number of pages pinned. This may be fewer than the number requested. If nr_pages is 0 or negative, returns 0. If no pages were pinned, returns -errno.

6.7.2 Memory Allocation Controls

Page mobility and placement hints

These flags provide hints about how mobile the page is. Pages with similar mobility are placed within the same pageblocks to minimise problems due to external fragmentation.

`__GFP_MOVABLE` (also a zone modifier) indicates that the page can be moved by page migration during memory compaction or can be reclaimed.

`__GFP_RECLAIMABLE` is used for slab allocations that specify SLAB_RECLAIM_ACCOUNT and whose pages can be freed via shrinkers.

__GFP_WRITE indicates the caller intends to dirty the page. Where possible, these pages will be spread between local zones to avoid all the dirty pages being in one zone (fair zone allocation policy).

__GFP_HARDWALL enforces the cpuset memory allocation policy.

__GFP_THISNODE forces the allocation to be satisfied from the requested node with no fallbacks or placement policy enforcements.

__GFP_ACCOUNT causes the allocation to be accounted to kmemcg.

Watermark modifiers -- controls access to emergency reserves

__GFP_HIGH indicates that the caller is high-priority and that granting the request is necessary before the system can make forward progress. For example creating an IO context to clean pages and requests from atomic context.

__GFP_MEMALLOC allows access to all memory. This should only be used when the caller guarantees the allocation will allow more memory to be freed very shortly e.g. process exiting or swapping. Users either should be the MM or co-ordinating closely with the VM (e.g. swap over NFS). Users of this flag have to be extremely careful to not deplete the reserve completely and implement a throttling mechanism which controls the consumption of the reserve based on the amount of freed memory. Usage of a pre-allocated pool (e.g. mempool) should be always considered before using this flag.

__GFP_NOMEMALLOC is used to explicitly forbid access to emergency reserves. This takes precedence over the __GFP_MEMALLOC flag if both are set.

Reclaim modifiers

Please note that all the following flags are only applicable to sleepable allocations (e.g. GFP_NOWAIT and GFP_ATOMIC will ignore them).

__GFP_IO can start physical IO.

__GFP_FS can call down to the low-level FS. Clearing the flag avoids the allocator recursing into the filesystem which might already be holding locks.

__GFP_DIRECT_RECLAIM indicates that the caller may enter direct reclaim. This flag can be cleared to avoid unnecessary delays when a fallback option is available.

__GFP_KSWAPD_RECLAIM indicates that the caller wants to wake kswapd when the low watermark is reached and have it reclaim pages until the high watermark is reached. A caller may wish to clear this flag when fallback options are available and the reclaim is likely to disrupt the system. The canonical example is THP allocation where a fallback is cheap but reclaim/compaction may cause indirect stalls.

__GFP_RECLAIM is shorthand to allow/forbid both direct and kswapd reclaim.

The default allocator behavior depends on the request size. We have a concept of so called costly allocations (with order > PAGE_ALLOC_COSTLY_ORDER). !costly allocations are too essential to fail so they are implicitly non-failing by default (with some exceptions like OOM victims might fail so the caller still has to check for failures) while costly requests try to be not disruptive and back off even without invoking the OOM killer. The following three modifiers might be used to override some of these implicit rules

__GFP_NORETRY: The VM implementation will try only very lightweight memory direct reclaim to get some memory under memory pressure (thus it can sleep). It will avoid disruptive actions like OOM killer. The caller must handle the failure which is quite likely to happen under heavy memory pressure. The flag is suitable when failure can easily be handled at small cost, such as reduced throughput

__GFP_RETRY_MAYFAIL: The VM implementation will retry memory reclaim procedures that have previously failed if there is some indication that progress has been made else where. It can wait for other tasks to attempt high level approaches to freeing memory such as compaction (which removes fragmentation) and page-out. There is still a definite limit to the number of retries, but it is a larger limit than with __GFP_NORETRY. Allocations with this flag may fail, but only when there is genuinely little unused memory. While these allocations do not directly trigger the OOM killer, their failure indicates that the system is likely to need to use the OOM killer soon. The caller must handle failure, but can reasonably do so by failing a higher-level request, or completing it only in a much less efficient manner. If the allocation does fail, and the caller is in a position to free some non-essential memory, doing so could benefit the system as a whole.

__GFP_NOFAIL: The VM implementation _must_ retry infinitely: the caller cannot handle allocation failures. The allocation could block indefinitely but will never return with failure. Testing for failure is pointless. New users should be evaluated carefully (and the flag should be used only when there is no reasonable failure policy) but it is definitely preferable to use the flag rather than opencode endless loop around allocator. Using this flag for costly allocations is _highly_ discouraged.

Useful GFP flag combinations

Useful GFP flag combinations that are commonly used. It is recommended that subsystems start with one of these combinations and then set/clear __GFP_FOO flags as necessary.

GFP_ATOMIC users can not sleep and need the allocation to succeed. A lower watermark is applied to allow access to "atomic reserves". The current implementation doesn't support NMI and few other strict non-preemptive contexts (e.g. raw_spin_lock). The same applies to GFP_NOWAIT.

GFP_KERNEL is typical for kernel-internal allocations. The caller requires ZONE_NORMAL or a lower zone for direct access but can direct reclaim.

GFP_KERNEL_ACCOUNT is the same as GFP_KERNEL, except the allocation is accounted to kmemcg.

GFP_NOWAIT is for kernel allocations that should not stall for direct reclaim, start physical IO or use any filesystem callback.

GFP_NOIO will use direct reclaim to discard clean pages or slab pages that do not require the starting of any physical IO. Please try to avoid using this flag directly and instead use memalloc_noio_{save,restore} to mark the whole scope which cannot perform any IO with a short explanation why. All allocation requests will inherit GFP_NOIO implicitly.

GFP_NOFS will use direct reclaim but will not use any filesystem interfaces. Please try to avoid using this flag directly and instead use memalloc_nofs_{save,restore} to mark the whole scope which cannot/shouldn't recurse into the FS layer with a short explanation why. All allocation requests will inherit GFP_NOFS implicitly.

GFP_USER is for userspace allocations that also need to be directly accessibly by the kernel or hardware. It is typically used by hardware for buffers that are mapped to userspace (e.g.

graphics) that hardware still must DMA to. cpuset limits are enforced for these allocations.

GFP_DMA exists for historical reasons and should be avoided where possible. The flags indicates that the caller requires that the lowest zone be used (ZONE_DMA or 16M on x86-64). Ideally, this would be removed but it would require careful auditing as some users really require it and others use the flag to avoid lowmem reserves in ZONE_DMA and treat the lowest zone as a type of emergency reserve.

GFP_DMA32 is similar to GFP_DMA except that the caller requires a 32-bit address. Note that kmalloc(..., GFP_DMA32) does not return DMA32 memory because the DMA32 kmalloc cache array is not implemented. (Reason: there is no such user in kernel).

GFP_HIGHUSER is for userspace allocations that may be mapped to userspace, do not need to be directly accessible by the kernel but that cannot move once in use. An example may be a hardware allocation that maps data directly into userspace but has no addressing limitations.

GFP_HIGHUSER_MOVABLE is for userspace allocations that the kernel does not need direct access to but can use kmap() when access is required. They are expected to be movable via page reclaim or page migration. Typically, pages on the LRU would also be allocated with GFP_HIGHUSER_MOVABLE.

GFP_TRANSHUGE and GFP_TRANSHUGE_LIGHT are used for THP allocations. They are compound allocations that will generally fail quickly if memory is not available and will not wake kswapd/kcompactd on failure. The _LIGHT version does not attempt reclaim/compaction at all and is by default used in page fault path, while the non-light is used by khugepaged.

6.7.3 The Slab Cache

size_t **ksize**(const void *objp)

Report actual allocation size of associated object

Parameters

const void *objp

Pointer returned from a prior kmalloc()-family allocation.

Description

This should not be used for writing beyond the originally requested allocation size. Either use krealloc() or round up the allocation size with *kmalloc_size_roundup()* prior to allocation. If this is used to access beyond the originally requested allocation size, UBSAN_BOUNDS and/or FORTIFY_SOURCE may trip, since they only know about the originally allocated size via the __alloc_size attribute.

void ***kmem_cache_alloc**(struct kmem_cache *cachep, gfp_t flags)

Allocate an object

Parameters

struct kmem_cache *cachep

The cache to allocate from.

gfp_t flags

See kmalloc().

Description

Allocate an object from this cache. See kmem_cache_zalloc() for a shortcut of adding __GFP_ZERO to flags.

Return

pointer to the new object or `NULL` in case of error

void *`kmalloc`(size_t size, gfp_t flags)
allocate kernel memory

Parameters

`size_t size`
how many bytes of memory are required.

`gfp_t flags`
describe the allocation context

Description

kmalloc is the normal method of allocating memory for objects smaller than page size in the kernel.

The allocated object address is aligned to at least ARCH_KMALLOC_MINALIGN bytes. For **size** of power of two bytes, the alignment is also guaranteed to be at least to the size.

The **flags** argument may be one of the GFP flags defined at include/linux/gfp_types.h and described at *Documentation/core-api/mm-api.rst*

The recommended usage of the **flags** is described at *Documentation/core-api/memory-allocation.rst*

Below is a brief outline of the most useful GFP flags

`GFP_KERNEL`
Allocate normal kernel ram. May sleep.

`GFP_NOWAIT`
Allocation will not sleep.

`GFP_ATOMIC`
Allocation will not sleep. May use emergency pools.

Also it is possible to set different flags by OR'ing in one or more of the following additional **flags**:

`__GFP_ZERO`
Zero the allocated memory before returning. Also see kzalloc().

`__GFP_HIGH`
This allocation has high priority and may use emergency pools.

`__GFP_NOFAIL`
Indicate that this allocation is in no way allowed to fail (think twice before using).

`__GFP_NORETRY`
If memory is not immediately available, then give up at once.

`__GFP_NOWARN`
If allocation fails, don't issue any warnings.

__GFP_RETRY_MAYFAIL
Try really hard to succeed the allocation but fail eventually.

void ***kmalloc_array**(size_t n, size_t size, gfp_t flags)
allocate memory for an array.

Parameters

size_t n
number of elements.

size_t size
element size.

gfp_t flags
the type of memory to allocate (see kmalloc).

void ***krealloc_array**(void *p, size_t new_n, size_t new_size, gfp_t flags)
reallocate memory for an array.

Parameters

void *p
pointer to the memory chunk to reallocate

size_t new_n
new number of elements to alloc

size_t new_size
new size of a single member of the array

gfp_t flags
the type of memory to allocate (see kmalloc)

void ***kcalloc**(size_t n, size_t size, gfp_t flags)
allocate memory for an array. The memory is set to zero.

Parameters

size_t n
number of elements.

size_t size
element size.

gfp_t flags
the type of memory to allocate (see kmalloc).

void ***kzalloc**(size_t size, gfp_t flags)
allocate memory. The memory is set to zero.

Parameters

size_t size
how many bytes of memory are required.

gfp_t flags
the type of memory to allocate (see kmalloc).

void *__kzalloc_node__(size_t size, gfp_t flags, int node)
allocate zeroed memory from a particular memory node.

Parameters

size_t size
how many bytes of memory are required.

gfp_t flags
the type of memory to allocate (see kmalloc).

int node
memory node from which to allocate

size_t **kmalloc_size_roundup**(size_t size)
Report allocation bucket size for the given size

Parameters

size_t size
Number of bytes to round up from.

Description

This returns the number of bytes that would be available in a kmalloc() allocation of **size** bytes. For example, a 126 byte request would be rounded up to the next sized kmalloc bucket, 128 bytes. (This is strictly for the general-purpose kmalloc()-based allocations, and is not for the pre-sized kmem_cache_alloc()-based allocations.)

Use this to kmalloc() the full bucket size ahead of time instead of using *ksize()* to query the size after an allocation.

void ***kmem_cache_alloc_node**(struct kmem_cache *cachep, gfp_t flags, int nodeid)
Allocate an object on the specified node

Parameters

struct kmem_cache *cachep
The cache to allocate from.

gfp_t flags
See kmalloc().

int nodeid
node number of the target node.

Description

Identical to kmem_cache_alloc but it will allocate memory on the given node, which can improve the performance for cpu bound structures.

Fallback to other node is possible if __GFP_THISNODE is not set.

Return

pointer to the new object or NULL in case of error

void **kmem_cache_free**(struct kmem_cache *cachep, void *objp)
Deallocate an object

Parameters

struct kmem_cache *cachep
The cache the allocation was from.

void *objp
The previously allocated object.

Description

Free an object which was previously allocated from this cache.

struct kmem_cache ***kmem_cache_create_usercopy**(const char *name, unsigned int size, unsigned int align, slab_flags_t flags, unsigned int useroffset, unsigned int usersize, void (*ctor)(void*))

Create a cache with a region suitable for copying to userspace

Parameters

const char *name
A string which is used in /proc/slabinfo to identify this cache.

unsigned int size
The size of objects to be created in this cache.

unsigned int align
The required alignment for the objects.

slab_flags_t flags
SLAB flags

unsigned int useroffset
Usercopy region offset

unsigned int usersize
Usercopy region size

void (*ctor)(void *)
A constructor for the objects.

Description

Cannot be called within a interrupt, but can be interrupted. The **ctor** is run when new pages are allocated by the cache.

The flags are

`SLAB_POISON` - Poison the slab with a known test pattern (a5a5a5a5) to catch references to uninitialised memory.

`SLAB_RED_ZONE` - Insert *Red* zones around the allocated memory to check for buffer overruns.

`SLAB_HWCACHE_ALIGN` - Align the objects in this cache to a hardware cacheline. This can be beneficial if you're counting cycles as closely as davem.

Return

a pointer to the cache on success, NULL on failure.

struct kmem_cache ***kmem_cache_create**(const char *name, unsigned int size, unsigned int align, slab_flags_t flags, void (*ctor)(void*))

Create a cache.

Parameters

const char *name
A string which is used in /proc/slabinfo to identify this cache.

unsigned int size
The size of objects to be created in this cache.

unsigned int align
The required alignment for the objects.

slab_flags_t flags
SLAB flags

void (*ctor)(void *)
A constructor for the objects.

Description

Cannot be called within a interrupt, but can be interrupted. The **ctor** is run when new pages are allocated by the cache.

The flags are

`SLAB_POISON` - Poison the slab with a known test pattern (a5a5a5a5) to catch references to uninitialised memory.

`SLAB_RED_ZONE` - Insert *Red* zones around the allocated memory to check for buffer overruns.

`SLAB_HWCACHE_ALIGN` - Align the objects in this cache to a hardware cacheline. This can be beneficial if you're counting cycles as closely as davem.

Return

a pointer to the cache on success, NULL on failure.

int **kmem_cache_shrink**(struct kmem_cache *cachep)
Shrink a cache.

Parameters

struct kmem_cache *cachep
The cache to shrink.

Description

Releases as many slabs as possible for a cache. To help debugging, a zero exit status indicates all slabs were released.

Return

`0` if all slabs were released, non-zero otherwise

bool **kmem_valid_obj**(void *object)
does the pointer reference a valid slab object?

Parameters

void *object
pointer to query.

Return

`true` if the pointer is to a not-yet-freed object from kmalloc() or kmem_cache_alloc(), either `true` or `false` if the pointer is to an already-freed object, and `false` otherwise.

void **kmem_dump_obj**(void *object)

Print available slab provenance information

Parameters

void *object
slab object for which to find provenance information.

Description

This function uses pr_cont(), so that the caller is expected to have printed out whatever preamble is appropriate. The provenance information depends on the type of object and on how much debugging is enabled. For a slab-cache object, the fact that it is a slab object is printed, and, if available, the slab name, return address, and stack trace from the allocation and last free path of that object.

This function will splat if passed a pointer to a non-slab object. If you are not sure what type of object you have, you should instead use mem_dump_obj().

void **kfree**(const void *object)

free previously allocated memory

Parameters

const void *object
pointer returned by kmalloc() or kmem_cache_alloc()

Description

If **object** is NULL, no operation is performed.

void ***krealloc**(const void *p, size_t new_size, gfp_t flags)

reallocate memory. The contents will remain unchanged.

Parameters

const void *p
object to reallocate memory for.

size_t new_size
how many bytes of memory are required.

gfp_t flags
the type of memory to allocate.

Description

The contents of the object pointed to are preserved up to the lesser of the new and old sizes (__GFP_ZERO flag is effectively ignored). If **p** is `NULL`, krealloc() behaves exactly like kmalloc(). If **new_size** is 0 and **p** is not a `NULL` pointer, the object pointed to is freed.

Return

pointer to the allocated memory or `NULL` in case of error void **kfree_sensitive**(const void *p)

Clear sensitive information in memory before freeing

Parameters

const void *p
object to free memory of

Description

The memory of the object **p** points to is zeroed before freed. If **p** is NULL, *kfree_sensitive()* does nothing.

Note

this function zeroes the whole allocated buffer which can be a good deal bigger than the requested buffer size passed to kmalloc(). So be careful when using this function in performance sensitive code.

void **kfree_const**(const void *x)

conditionally free memory

Parameters

const void *x
pointer to the memory

Description

Function calls kfree only if **x** is not in .rodata section.

void ***kvmalloc_node**(size_t size, gfp_t flags, int node)

attempt to allocate physically contiguous memory, but upon failure, fall back to non-contiguous (vmalloc) allocation.

Parameters

size_t size
size of the request.

gfp_t flags
gfp mask for the allocation - must be compatible (superset) with GFP_KERNEL.

int node
numa node to allocate from

Description

Uses kmalloc to get the memory but if the allocation fails then falls back to the vmalloc allocator. Use kvfree for freeing the memory.

GFP_NOWAIT and GFP_ATOMIC are not supported, neither is the __GFP_NORETRY modifier. __GFP_RETRY_MAYFAIL is supported, and it should be used only if kmalloc is preferable to the vmalloc fallback, due to visible performance drawbacks.

Return

pointer to the allocated memory of NULL in case of failure

void **kvfree**(const void *addr)

Free memory.

Parameters

const void *addr
Pointer to allocated memory.

Description

kvfree frees memory allocated by any of vmalloc(), kmalloc() or kvmalloc(). It is slightly more efficient to use kfree() or *vfree()* if you are certain that you know which one to use.

Context

Either preemptible task context or not-NMI interrupt.

6.7.4 Virtually Contiguous Mappings

void **vm_unmap_aliases**(void)
unmap outstanding lazy aliases in the vmap layer

Parameters

void
no arguments

Description

The vmap/vmalloc layer lazily flushes kernel virtual mappings primarily to amortize TLB flushing overheads. What this means is that any page you have now, may, in a former life, have been mapped into kernel virtual address by the vmap layer and so there might be some CPUs with TLB entries still referencing that page (additional to the regular 1:1 kernel mapping).

vm_unmap_aliases flushes all such lazy mappings. After it returns, we can be sure that none of the pages we have control over will have any aliases from the vmap layer.

void **vm_unmap_ram**(const void *mem, unsigned int count)
unmap linear kernel address space set up by vm_map_ram

Parameters

const void *mem
the pointer returned by vm_map_ram

unsigned int count
the count passed to that vm_map_ram call (cannot unmap partial)

void ***vm_map_ram**(struct page **pages, unsigned int count, int node)
map pages linearly into kernel virtual address (vmalloc space)

Parameters

struct page **pages
an array of pointers to the pages to be mapped

unsigned int count
number of pages

int node
prefer to allocate data structures on this node

Description

If you use this function for less than VMAP_MAX_ALLOC pages, it could be faster than vmap so it's good. But if you mix long-life and short-life objects with *vm_map_ram()*, it could consume lots of address space through fragmentation (especially on a 32bit machine). You could see failures in the end. Please use this function for short-lived objects.

Return

a pointer to the address that has been mapped, or `NULL` on failure

void **vfree**(const void *addr)

Release memory allocated by vmalloc()

Parameters

const void *addr
Memory base address

Description

Free the virtually continuous memory area starting at **addr**, as obtained from one of the vmalloc() family of APIs. This will usually also free the physical memory underlying the virtual allocation, but that memory is reference counted, so it will not be freed until the last user goes away.

If **addr** is NULL, no operation is performed.

Context

May sleep if called *not* from interrupt context. Must not be called in NMI context (strictly speaking, it could be if we have CONFIG_ARCH_HAVE_NMI_SAFE_CMPXCHG, but making the calling conventions for *vfree()* arch-dependent would be a really bad idea).

void **vunmap**(const void *addr)

release virtual mapping obtained by vmap()

Parameters

const void *addr
memory base address

Description

Free the virtually contiguous memory area starting at **addr**, which was created from the page array passed to vmap().

Must not be called in interrupt context.

void ***vmap**(struct page **pages, unsigned int count, unsigned long flags, pgprot_t prot)

map an array of pages into virtually contiguous space

Parameters

struct page **pages
array of page pointers

unsigned int count
number of pages to map

unsigned long flags
vm_area->flags

pgprot_t prot
page protection for the mapping

Description

Maps **count** pages from **pages** into contiguous kernel virtual space. If **flags** contains VM_MAP_PUT_PAGES the ownership of the pages array itself (which must be kmalloc or vmalloc memory) and one reference per pages in it are transferred from the caller to vmap(), and will be freed / dropped when *vfree()* is called on the return value.

Return

the address of the area or NULL on failure

void ***vmap_pfn**(unsigned long *pfns, unsigned int count, pgprot_t prot)
map an array of PFNs into virtually contiguous space

Parameters

unsigned long *pfns
array of PFNs

unsigned int count
number of pages to map

pgprot_t prot
page protection for the mapping

Description

Maps **count** PFNs from **pfns** into contiguous kernel virtual space and returns the start address of the mapping.

void ***__vmalloc_node**(unsigned long size, unsigned long align, gfp_t gfp_mask, int node, const void *caller)
allocate virtually contiguous memory

Parameters

unsigned long size
allocation size

unsigned long align
desired alignment

gfp_t gfp_mask
flags for the page level allocator

int node
node to use for allocation or NUMA_NO_NODE

const void *caller
caller's return address

Description

Allocate enough pages to cover **size** from the page level allocator with **gfp_mask** flags. Map them into contiguous kernel virtual space.

Reclaim modifiers in **gfp_mask** - __GFP_NORETRY, __GFP_RETRY_MAYFAIL and __GFP_NOFAIL are not supported

Any use of gfp flags outside of GFP_KERNEL should be consulted with mm people.

Return

pointer to the allocated memory or `NULL` on error

void *`vmalloc`(unsigned long size)
    allocate virtually contiguous memory

Parameters

`unsigned long size`
    allocation size

Description

Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space.

For tight control over page level allocator and protection flags use __vmalloc() instead.

Return

pointer to the allocated memory or `NULL` on error

void *`vmalloc_huge`(unsigned long size, gfp_t gfp_mask)
    allocate virtually contiguous memory, allow huge pages

Parameters

`unsigned long size`
    allocation size

`gfp_t gfp_mask`
    flags for the page level allocator

Description

Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space. If **size** is greater than or equal to PMD_SIZE, allow using huge pages for the memory

Return

pointer to the allocated memory or `NULL` on error

void *`vzalloc`(unsigned long size)
    allocate virtually contiguous memory with zero fill

Parameters

`unsigned long size`
    allocation size

Description

Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space. The memory allocated is set to zero.

For tight control over page level allocator and protection flags use __vmalloc() instead.

Return

pointer to the allocated memory or `NULL` on error void *`vmalloc_user`(unsigned long size)

allocate zeroed virtually contiguous memory for userspace

Parameters

`unsigned long size`
allocation size

Description

The resulting memory area is zeroed so it can be mapped to userspace without leaking data.

Return

pointer to the allocated memory or `NULL` on error

void *`vmalloc_node`(unsigned long size, int node)

allocate memory on a specific node

Parameters

`unsigned long size`
allocation size

`int node`
numa node

Description

Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space.

For tight control over page level allocator and protection flags use __vmalloc() instead.

Return

pointer to the allocated memory or `NULL` on error

void *`vzalloc_node`(unsigned long size, int node)

allocate memory on a specific node with zero fill

Parameters

`unsigned long size`
allocation size

`int node`
numa node

Description

Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space. The memory allocated is set to zero.

Return

pointer to the allocated memory or `NULL` on error

void *`vmalloc_32`(unsigned long size)

allocate virtually contiguous memory (32bit addressable)

Parameters

unsigned long size
allocation size

Description

Allocate enough 32bit PA addressable pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space.

Return

pointer to the allocated memory or NULL on error

void ***vmalloc_32_user**(unsigned long size)
allocate zeroed virtually contiguous 32bit memory

Parameters

unsigned long size
allocation size

Description

The resulting memory area is 32bit addressable and zeroed so it can be mapped to userspace without leaking data.

Return

pointer to the allocated memory or NULL on error

int **remap_vmalloc_range**(struct vm_area_struct *vma, void *addr, unsigned long pgoff)
map vmalloc pages to userspace

Parameters

struct vm_area_struct *vma
vma to cover (map full range of vma)

void *addr
vmalloc memory

unsigned long pgoff
number of pages into addr before first page to map

Return

0 for success, -Exxx on failure

Description

This function checks that addr is a valid vmalloc'ed area, and that it is big enough to cover the vma. Will return failure if that criteria isn't met.

Similar to *remap_pfn_range()* (see mm/memory.c)

6.7.5 File Mapping and Page Cache

Filemap

int **filemap_fdatawrite_wbc**(struct address_space *mapping, struct writeback_control *wbc)
start writeback on mapping dirty pages in range

Parameters

struct address_space *mapping
address space structure to write

struct writeback_control *wbc
the writeback_control controlling the writeout

Description

Call writepages on the mapping using the provided wbc to control the writeout.

Return

`0` on success, negative error code otherwise.

int **filemap_flush**(struct address_space *mapping)
mostly a non-blocking flush

Parameters

struct address_space *mapping
target address_space

Description

This is a mostly non-blocking flush. Not suitable for data-integrity purposes - I/O may not be started against all dirty pages.

Return

`0` on success, negative error code otherwise.

bool **filemap_range_has_page**(struct address_space *mapping, loff_t start_byte, loff_t end_byte)
check if a page exists in range.

Parameters

struct address_space *mapping
address space within which to check

loff_t start_byte
offset in bytes where the range starts

loff_t end_byte
offset in bytes where the range ends (inclusive)

Description

Find at least one page in the range supplied, usually used to check if direct writing in this range will trigger a writeback.

Return

`true` if at least one page exists in the specified range, `false` otherwise.

int **filemap_fdatawait_range**(struct address_space *mapping, loff_t start_byte, loff_t end_byte)

wait for writeback to complete

Parameters

struct address_space *mapping
address space structure to wait for

loff_t start_byte
offset in bytes where the range starts

loff_t end_byte
offset in bytes where the range ends (inclusive)

Description

Walk the list of under-writeback pages of the given address space in the given range and wait for all of them. Check error status of the address space and return it.

Since the error status of the address space is cleared by this function, callers are responsible for checking the return value and handling and/or reporting the error.

Return

error status of the address space.

int **filemap_fdatawait_range_keep_errors**(struct address_space *mapping, loff_t start_byte, loff_t end_byte)

wait for writeback to complete

Parameters

struct address_space *mapping
address space structure to wait for

loff_t start_byte
offset in bytes where the range starts

loff_t end_byte
offset in bytes where the range ends (inclusive)

Description

Walk the list of under-writeback pages of the given address space in the given range and wait for all of them. Unlike *filemap_fdatawait_range()*, this function does not clear error status of the address space.

Use this function if callers don't handle errors themselves. Expected call sites are system-wide / filesystem-wide data flushers: e.g. sync(2), fsfreeze(8)

int **file_fdatawait_range**(struct *file* *file, loff_t start_byte, loff_t end_byte)

wait for writeback to complete

Parameters

struct file *file
file pointing to address space structure to wait for

loff_t start_byte
offset in bytes where the range starts

loff_t end_byte
offset in bytes where the range ends (inclusive)

Description

Walk the list of under-writeback pages of the address space that file refers to, in the given range and wait for all of them. Check error status of the address space vs. the file->f_wb_err cursor and return it.

Since the error status of the file is advanced by this function, callers are responsible for checking the return value and handling and/or reporting the error.

Return

error status of the address space vs. the file->f_wb_err cursor.

int **filemap_fdatawait_keep_errors**(struct address_space *mapping)
wait for writeback without clearing errors

Parameters

struct address_space *mapping
address space structure to wait for

Description

Walk the list of under-writeback pages of the given address space and wait for all of them. Unlike filemap_fdatawait(), this function does not clear error status of the address space.

Use this function if callers don't handle errors themselves. Expected call sites are system-wide / filesystem-wide data flushers: e.g. sync(2), fsfreeze(8)

Return

error status of the address space.

int **filemap_write_and_wait_range**(struct address_space *mapping, loff_t lstart, loff_t lend)
write out & wait on a file range

Parameters

struct address_space *mapping
the address_space for the pages

loff_t lstart
offset in bytes where the range starts

loff_t lend
offset in bytes where the range ends (inclusive)

Description

Write out and wait upon file offsets lstart->lend, inclusive.

Note that **lend** is inclusive (describes the last byte to be written) so that this function can be used to write to the very end-of-file (end = -1).

Return

error status of the address space.

int **file_check_and_advance_wb_err**(struct *file* *file)

report wb error (if any) that was previously and advance wb_err to current one

Parameters

struct file *file

struct file on which the error is being reported

Description

When userland calls fsync (or something like nfsd does the equivalent), we want to report any writeback errors that occurred since the last fsync (or since the file was opened if there haven't been any).

Grab the wb_err from the mapping. If it matches what we have in the file, then just quickly return 0. The file is all caught up.

If it doesn't match, then take the mapping value, set the "seen" flag in it and try to swap it into place. If it works, or another task beat us to it with the new value, then update the f_wb_err and return the error portion. The error at this point must be reported via proper channels (a'la fsync, or NFS COMMIT operation, etc.).

While we handle mapping->wb_err with atomic operations, the f_wb_err value is protected by the f_lock since we must ensure that it reflects the latest value swapped in for this file descriptor.

Return

`0` on success, negative error code otherwise.

int **file_write_and_wait_range**(struct *file* *file, loff_t lstart, loff_t lend)

write out & wait on a file range

Parameters

struct file *file

file pointing to address_space with pages

loff_t lstart

offset in bytes where the range starts

loff_t lend

offset in bytes where the range ends (inclusive)

Description

Write out and wait upon file offsets lstart->lend, inclusive.

Note that **lend** is inclusive (describes the last byte to be written) so that this function can be used to write to the very end-of-file (end = -1).

After writing out and waiting on the data, we check and advance the f_wb_err cursor to the latest value, and return any errors detected there.

Return

`0` on success, negative error code otherwise.

void **replace_page_cache_folio**(struct *folio* *old, struct *folio* *new)

replace a pagecache folio with a new one

Parameters

struct folio *old
folio to be replaced

struct folio *new
folio to replace with

Description

This function replaces a folio in the pagecache with a new one. On success it acquires the pagecache reference for the new folio and drops it for the old folio. Both the old and new folios must be locked. This function does not add the new folio to the LRU, the caller must do that.

The remove + add is atomic. This function cannot fail.

void **folio_add_wait_queue**(struct *folio* *folio, wait_queue_entry_t *waiter)
Add an arbitrary waiter to a folio's wait queue

Parameters

struct folio *folio
Folio defining the wait queue of interest

wait_queue_entry_t *waiter
Waiter to add to the queue

Description

Add an arbitrary **waiter** to the wait queue for the nominated **folio**.

void **folio_unlock**(struct *folio* *folio)
Unlock a locked folio.

Parameters

struct folio *folio
The folio.

Description

Unlocks the folio and wakes up any thread sleeping on the page lock.

Context

May be called from interrupt or process context. May not be called from NMI context.

void **folio_end_private_2**(struct *folio* *folio)
Clear PG_private_2 and wake any waiters.

Parameters

struct folio *folio
The folio.

Description

Clear the PG_private_2 bit on a folio and wake up any sleepers waiting for it. The folio reference held for PG_private_2 being set is released.

This is, for example, used when a netfs folio is being written to a local disk cache, thereby allowing writes to the cache for the same folio to be serialised.

void **folio_wait_private_2**(struct *folio* *folio)

Wait for PG_private_2 to be cleared on a folio.

Parameters

struct folio *folio

The folio to wait on.

Description

Wait for PG_private_2 (aka PG_fscache) to be cleared on a folio.

int **folio_wait_private_2_killable**(struct *folio* *folio)

Wait for PG_private_2 to be cleared on a folio.

Parameters

struct folio *folio

The folio to wait on.

Description

Wait for PG_private_2 (aka PG_fscache) to be cleared on a folio or until a fatal signal is received by the calling task.

Return

- 0 if successful.
- -EINTR if a fatal signal was encountered.

void **folio_end_writeback**(struct *folio* *folio)

End writeback against a folio.

Parameters

struct folio *folio

The folio.

void **__folio_lock**(struct *folio* *folio)

Get a lock on the folio, assuming we need to sleep to get it.

Parameters

struct folio *folio

The folio to lock

pgoff_t **page_cache_next_miss**(struct address_space *mapping, pgoff_t index, unsigned long max_scan)

Find the next gap in the page cache.

Parameters

struct address_space *mapping

Mapping.

pgoff_t index

Index.

unsigned long max_scan

Maximum range to search.

Description

Search the range [index, min(index + max_scan - 1, ULONG_MAX)] for the gap with the lowest index.

This function may be called under the rcu_read_lock. However, this will not atomically search a snapshot of the cache at a single point in time. For example, if a gap is created at index 5, then subsequently a gap is created at index 10, page_cache_next_miss covering both indices may return 10 if called under the rcu_read_lock.

Return

The index of the gap if found, otherwise an index outside the range specified (in which case 'return - index >= max_scan' will be true). In the rare case of index wrap-around, 0 will be returned.

pgoff_t **page_cache_prev_miss**(struct address_space *mapping, pgoff_t index, unsigned long max_scan)

Find the previous gap in the page cache.

Parameters

struct address_space *mapping
Mapping.

pgoff_t index
Index.

unsigned long max_scan
Maximum range to search.

Description

Search the range [max(index - max_scan + 1, 0), index] for the gap with the highest index.

This function may be called under the rcu_read_lock. However, this will not atomically search a snapshot of the cache at a single point in time. For example, if a gap is created at index 10, then subsequently a gap is created at index 5, *page_cache_prev_miss()* covering both indices may return 5 if called under the rcu_read_lock.

Return

The index of the gap if found, otherwise an index outside the range specified (in which case 'index - return >= max_scan' will be true). In the rare case of wrap-around, ULONG_MAX will be returned.

struct *folio* ***__filemap_get_folio**(struct address_space *mapping, pgoff_t index, *fgf_t* fgp_flags, gfp_t gfp)

Find and get a reference to a folio.

Parameters

struct address_space *mapping
The address_space to search.

pgoff_t index
The page index.

fgf_t fgp_flags
FGP flags modify how the folio is returned.

gfp_t gfp
Memory allocation flags to use if FGP_CREAT is specified.

Description

Looks up the page cache entry at **mapping** & **index**.

If FGP_LOCK or FGP_CREAT are specified then the function may sleep even if the GFP flags specified for FGP_CREAT are atomic.

If this function returns a folio, it is returned with an increased refcount.

Return

The found folio or an ERR_PTR() otherwise.

unsigned **filemap_get_folios**(struct address_space *mapping, pgoff_t *start, pgoff_t end, struct folio_batch *fbatch)

Get a batch of folios

Parameters

struct address_space *mapping
The address_space to search

pgoff_t *start
The starting page index

pgoff_t end
The final page index (inclusive)

struct folio_batch *fbatch
The batch to fill.

Description

Search for and return a batch of folios in the mapping starting at index **start** and up to index **end** (inclusive). The folios are returned in **fbatch** with an elevated reference count.

The first folio may start before **start**; if it does, it will contain **start**. The final folio may extend beyond **end**; if it does, it will contain **end**. The folios have ascending indices. There may be gaps between the folios if there are indices which have no folio in the page cache. If folios are added to or removed from the page cache while this is running, they may or may not be found by this call.

Return

The number of folios which were found. We also update **start** to index the next folio for the traversal.

unsigned **filemap_get_folios_contig**(struct address_space *mapping, pgoff_t *start, pgoff_t end, struct folio_batch *fbatch)

Get a batch of contiguous folios

Parameters

struct address_space *mapping
The address_space to search

pgoff_t *start
The starting page index

pgoff_t end
The final page index (inclusive)

struct folio_batch *fbatch
The batch to fill

Description

filemap_get_folios_contig() works exactly like *filemap_get_folios()*, except the returned folios are guaranteed to be contiguous. This may not return all contiguous folios if the batch gets filled up.

Return

The number of folios found. Also update **start** to be positioned for traversal of the next folio.

unsigned **filemap_get_folios_tag**(struct address_space *mapping, pgoff_t *start, pgoff_t end, xa_mark_t tag, struct folio_batch *fbatch)

Get a batch of folios matching **tag**

Parameters

struct address_space *mapping
The address_space to search

pgoff_t *start
The starting page index

pgoff_t end
The final page index (inclusive)

xa_mark_t tag
The tag index

struct folio_batch *fbatch
The batch to fill

Description

Same as *filemap_get_folios()*, but only returning folios tagged with **tag**.

Return

The number of folios found. Also update **start** to index the next folio for traversal.

ssize_t **filemap_read**(struct kiocb *iocb, struct iov_iter *iter, ssize_t already_read)

Read data from the page cache.

Parameters

struct kiocb *iocb
The iocb to read.

struct iov_iter *iter
Destination for the data.

ssize_t already_read
Number of bytes already read by the caller.

Description

Copies data from the page cache. If the data is not currently present, uses the readahead and read_folio address_space operations to fetch it.

Return

Total number of bytes copied, including those already read by the caller. If an error happens before any bytes are copied, returns a negative error number.

ssize_t **generic_file_read_iter**(struct kiocb *iocb, struct iov_iter *iter)

generic filesystem read routine

Parameters

struct kiocb *iocb
kernel I/O control block

struct iov_iter *iter
destination for the data read

Description

This is the "read_iter()" routine for all filesystems that can use the page cache directly.

The IOCB_NOWAIT flag in iocb->ki_flags indicates that -EAGAIN shall be returned when no data can be read without waiting for I/O requests to complete; it doesn't prevent readahead.

The IOCB_NOIO flag in iocb->ki_flags indicates that no new I/O requests shall be made for the read or for readahead. When no data can be read, -EAGAIN shall be returned. When readahead would be triggered, a partial, possibly empty read shall be returned.

Return

- number of bytes copied, even for partial reads
- negative error code (or 0 if IOCB_NOIO) if nothing was read

ssize_t **filemap_splice_read**(struct file *in, loff_t *ppos, struct pipe_inode_info *pipe, size_t len, unsigned int flags)

Splice data from a file's pagecache into a pipe

Parameters

struct file *in
The file to read from

loff_t *ppos
Pointer to the file position to read from

struct pipe_inode_info *pipe
The pipe to splice into

size_t len
The amount to splice

unsigned int flags
The SPLICE_F_* flags

Description

This function gets folios from a file's pagecache and splices them into the pipe. Readahead will be called as necessary to fill more folios. This may be used for blockdevs also.

Return

On success, the number of bytes read will be returned and ***ppos** will be updated if appropriate; 0 will be returned if there is no more data to be read; -EAGAIN will be returned if the pipe had no space, and some other negative error code will be returned on error. A short read may occur if the pipe has insufficient space, we reach the end of the data or we hit a hole.

vm_fault_t **filemap_fault**(struct vm_fault *vmf)

read in file data for page fault handling

Parameters

struct vm_fault *vmf
struct vm_fault containing details of the fault

Description

filemap_fault() is invoked via the vma operations vector for a mapped memory region to read in file data during a page fault.

The goto's are kind of ugly, but this streamlines the normal case of having it in the page cache, and handles the special cases reasonably without having a lot of duplicated code.

vma->vm_mm->mmap_lock must be held on entry.

If our return value has VM_FAULT_RETRY set, it's because the mmap_lock may be dropped before doing I/O or by lock_folio_maybe_drop_mmap().

If our return value does not have VM_FAULT_RETRY set, the mmap_lock has not been released.

We never return with VM_FAULT_RETRY and a bit from VM_FAULT_ERROR set.

Return

bitwise-OR of `VM_FAULT_` codes.

struct *folio* ***read_cache_folio**(struct address_space *mapping, pgoff_t index, filler_t filler, struct *file* *file)

Read into page cache, fill it if needed.

Parameters

struct address_space *mapping
The address_space to read from.

pgoff_t index
The index to read.

filler_t filler
Function to perform the read, or NULL to use aops->read_folio().

struct file *file
Passed to filler function, may be NULL if not required.

Description

Read one page into the page cache. If it succeeds, the folio returned will contain **index**, but it may not be the first page of the folio.

If the filler function returns an error, it will be returned to the caller.

Context

May sleep. Expects mapping->invalidate_lock to be held.

Return

An uptodate folio on success, ERR_PTR() on failure.

struct *folio* ***mapping_read_folio_gfp***(struct address_space *mapping, pgoff_t index, gfp_t gfp)

Read into page cache, using specified allocation flags.

Parameters

struct address_space *mapping
The address_space for the folio.

pgoff_t index
The index that the allocated folio will contain.

gfp_t gfp
The page allocator flags to use if allocating.

Description

This is the same as "read_cache_folio(mapping, index, NULL, NULL)", but with any new memory allocations done using the specified allocation flags.

The most likely error from this function is EIO, but ENOMEM is possible and so is EINTR. If ->read_folio returns another error, that will be returned to the caller.

The function expects mapping->invalidate_lock to be already held.

Return

Uptodate folio on success, ERR_PTR() on failure.

struct page ***read_cache_page_gfp***(struct address_space *mapping, pgoff_t index, gfp_t gfp)

read into page cache, using specified page allocation flags.

Parameters

struct address_space *mapping
the page's address_space

pgoff_t index
the page index

gfp_t gfp
the page allocator flags to use if allocating

Description

This is the same as "read_mapping_page(mapping, index, NULL)", but with any new page allocations done using the specified allocation flags.

If the page does not get brought uptodate, return -EIO.

The function expects mapping->invalidate_lock to be already held.

Return

up to date page on success, ERR_PTR() on failure.

ssize_t **__generic_file_write_iter**(struct kiocb *iocb, struct iov_iter *from)
write data to a file

Parameters

struct kiocb *iocb
IO state structure (file, offset, etc.)

struct iov_iter *from
iov_iter with data to write

Description

This function does all the work needed for actually writing data to a file. It does all basic checks, removes SUID from the file, updates modification times and calls proper subroutines depending on whether we do direct IO or a standard buffered write.

It expects i_rwsem to be grabbed unless we work on a block device or similar object which does not need locking at all.

This function does *not* take care of syncing data in case of O_SYNC write. A caller has to handle it. This is mainly due to the fact that we want to avoid syncing under i_rwsem.

Return

- number of bytes written, even for truncated writes
- negative error code if no data has been written at all

ssize_t **generic_file_write_iter**(struct kiocb *iocb, struct iov_iter *from)
write data to a file

Parameters

struct kiocb *iocb
IO state structure

struct iov_iter *from
iov_iter with data to write

Description

This is a wrapper around *__generic_file_write_iter()* to be used by most filesystems. It takes care of syncing the file in case of O_SYNC file and acquires i_rwsem as needed.

Return

- negative error code if no data has been written at all of vfs_fsync_range() failed for a synchronous write
- number of bytes written, even for truncated writes

bool **filemap_release_folio**(struct *folio* *folio, gfp_t gfp)
Release fs-specific metadata on a folio.

Parameters

struct folio *folio
The folio which the kernel is trying to free.

gfp_t gfp
Memory allocation flags (and I/O mode).

Description

The address_space is trying to release any data attached to a folio (presumably at folio->private).

This will also be called if the private_2 flag is set on a page, indicating that the folio has other metadata associated with it.

The **gfp** argument specifies whether I/O may be performed to release this page (__GFP_IO), and whether the call may block (__GFP_RECLAIM & __GFP_FS).

Return

`true` if the release was successful, otherwise `false`.

Readahead

Readahead is used to read content into the page cache before it is explicitly requested by the application. Readahead only ever attempts to read folios that are not yet in the page cache. If a folio is present but not up-to-date, readahead will not try to read it. In that case a simple ->read_folio() will be requested.

Readahead is triggered when an application read request (whether a system call or a page fault) finds that the requested folio is not in the page cache, or that it is in the page cache and has the readahead flag set. This flag indicates that the folio was read as part of a previous readahead request and now that it has been accessed, it is time for the next readahead.

Each readahead request is partly synchronous read, and partly async readahead. This is reflected in the struct file_ra_state which contains ->size being the total number of pages, and ->async_size which is the number of pages in the async section. The readahead flag will be set on the first folio in this async section to trigger a subsequent readahead. Once a series of sequential reads has been established, there should be no need for a synchronous component and all readahead request will be fully asynchronous.

When either of the triggers causes a readahead, three numbers need to be determined: the start of the region to read, the size of the region, and the size of the async tail.

The start of the region is simply the first page address at or after the accessed address, which is not currently populated in the page cache. This is found with a simple search in the page cache.

The size of the async tail is determined by subtracting the size that was explicitly requested from the determined request size, unless this would be less than zero - then zero is used. NOTE THIS CALCULATION IS WRONG WHEN THE START OF THE REGION IS NOT THE ACCESSED PAGE. ALSO THIS CALCULATION IS NOT USED CONSISTENTLY.

The size of the region is normally determined from the size of the previous readahead which loaded the preceding pages. This may be discovered from the struct file_ra_state for simple sequential reads, or from examining the state of the page cache when multiple sequential reads are interleaved. Specifically: where the readahead was triggered by the readahead flag, the size of the previous readahead is assumed to be the number of pages from the triggering page to the start of the new readahead. In these cases, the size of the previous readahead is scaled, often doubled, for the new readahead, though see get_next_ra_size() for details.

If the size of the previous read cannot be determined, the number of preceding pages in the page cache is used to estimate the size of a previous read. This estimate could easily be misled by random reads being coincidentally adjacent, so it is ignored unless it is larger than the current request, and it is not scaled up, unless it is at the start of file.

In general readahead is accelerated at the start of the file, as reads from there are often sequential. There are other minor adjustments to the readahead size in various special cases and these are best discovered by reading the code.

The above calculation, based on the previous readahead size, determines the size of the readahead, to which any requested read size may be added.

Readahead requests are sent to the filesystem using the ->readahead() address space operation, for which mpage_readahead() is a canonical implementation. ->readahead() should normally initiate reads on all folios, but may fail to read any or all folios without causing an I/O error. The page cache reading code will issue a ->read_folio() request for any folio which ->readahead() did not read, and only an error from this will be final.

->readahead() will generally call *readahead_folio()* repeatedly to get each folio from those prepared for readahead. It may fail to read a folio by:

- not calling *readahead_folio()* sufficiently many times, effectively ignoring some folios, as might be appropriate if the path to storage is congested.
- failing to actually submit a read request for a given folio, possibly due to insufficient resources, or
- getting an error during subsequent processing of a request.

In the last two cases, the folio should be unlocked by the filesystem to indicate that the read attempt has failed. In the first case the folio will be unlocked by the VFS.

Those folios not in the final `async_size` of the request should be considered to be important and ->readahead() should not fail them due to congestion or temporary resource unavailability, but should wait for necessary resources (e.g. memory or indexing information) to become available. Folios in the final `async_size` may be considered less urgent and failure to read them is more acceptable. In this case it is best to use filemap_remove_folio() to remove the folios from the page cache as is automatically done for folios that were not fetched with *readahead_folio()*. This will allow a subsequent synchronous readahead request to try them again. If they are left in the page cache, then they will be read individually using ->read_folio() which may be less efficient.

void **page_cache_ra_unbounded**(struct *readahead_control* *ractl, unsigned long nr_to_read, unsigned long lookahead_size)

Start unchecked readahead.

Parameters

struct readahead_control *ractl
Readahead control.

unsigned long nr_to_read
The number of pages to read.

unsigned long lookahead_size
Where to start the next readahead.

Description

This function is for filesystems to call when they want to start readahead beyond a file's stated i_size. This is almost certainly not the function you want to call. Use *page_cache_async_readahead()* or *page_cache_sync_readahead()* instead.

Context

File is referenced by caller. Mutexes may be held by caller. May sleep, but will not reenter filesystem to reclaim memory.

void **readahead_expand**(struct *readahead_control* *ractl, loff_t new_start, size_t new_len)

Expand a readahead request

Parameters

struct readahead_control *ractl
The request to be expanded

loff_t new_start
The revised start

size_t new_len
The revised size of the request

Description

Attempt to expand a readahead request outwards from the current size to the specified size by inserting locked pages before and after the current window to increase the size to the new window. This may involve the insertion of THPs, in which case the window may get expanded even beyond what was requested.

The algorithm will stop if it encounters a conflicting page already in the pagecache and leave a smaller expansion than requested.

The caller must check for this by examining the revised **ractl** object for a different expansion than was requested.

Writeback

int **balance_dirty_pages_ratelimited_flags**(struct address_space *mapping, unsigned int flags)

Balance dirty memory state.

Parameters

struct address_space *mapping
address_space which was dirtied.

unsigned int flags
BDP flags.

Description

Processes which are dirtying memory should call in here once for each page which was newly dirtied. The function will periodically check the system's dirty state and will initiate writeback if needed.

See *balance_dirty_pages_ratelimited()* for details.

Return

If **flags** contains BDP_ASYNC, it may return -EAGAIN to indicate that memory is out of balance and the caller must wait for I/O to complete. Otherwise, it will return 0 to indicate that either memory was already in balance, or it was able to sleep until the amount of dirty memory returned to balance.

void **balance_dirty_pages_ratelimited**(struct address_space *mapping)

balance dirty memory state.

Parameters

struct address_space *mapping
address_space which was dirtied.

Description

Processes which are dirtying memory should call in here once for each page which was newly dirtied. The function will periodically check the system's dirty state and will initiate writeback if needed.

Once we're over the dirty memory limit we decrease the ratelimiting by a lot, to prevent individual processes from overshooting the limit by (ratelimit_pages) each.

void **tag_pages_for_writeback**(struct address_space *mapping, pgoff_t start, pgoff_t end)

tag pages to be written by write_cache_pages

Parameters

struct address_space *mapping
address space structure to write

pgoff_t start
starting page index

pgoff_t end
ending page index (inclusive)

Description

This function scans the page range from **start** to **end** (inclusive) and tags all pages that have DIRTY tag set with a special TOWRITE tag. The idea is that write_cache_pages (or whoever calls this function) will then use TOWRITE tag to identify pages eligible for writeback. This mechanism is used to avoid livelocking of writeback by a process steadily creating new dirty pages in the file (thus it is important for this function to be quick so that it can tag pages faster than a dirtying process can create them).

int **write_cache_pages**(struct address_space *mapping, struct writeback_control *wbc, writepage_t writepage, void *data)

walk the list of dirty pages of the given address space and write all of them.

Parameters

struct address_space *mapping
address space structure to write

struct writeback_control *wbc
subtract the number of written pages from ***wbc->nr_to_write**

writepage_t writepage
function called for each page

void *data
data passed to writepage function

Description

If a page is already under I/O, *write_cache_pages()* skips it, even if it's dirty. This is desirable behaviour for memory-cleaning writeback, but it is INCORRECT for data-integrity system calls such as fsync(). fsync() and msync() need to guarantee that all the data which was dirty at the time the call was made get new I/O started against them. If wbc->sync_mode is WB_SYNC_ALL then we were called for data integrity and we must wait for existing IO to complete.

To avoid livelocks (when other process dirties new pages), we first tag pages which should be written back with TOWRITE tag and only then start writing them. For data-integrity sync we have to be careful so that we do not miss some pages (e.g., because some other process has cleared TOWRITE tag we set). The rule we follow is that TOWRITE tag can be cleared only by the process clearing the DIRTY tag (and submitting the page for IO).

To avoid deadlocks between range_cyclic writeback and callers that hold pages in PageWriteback to aggregate IO until *write_cache_pages()* returns, we do not loop back to the start of the file. Doing so causes a page lock/page writeback access order inversion - we should only ever lock multiple pages in ascending page->index order, and looping back to the start of the file violates that rule and causes deadlocks.

Return

`0` on success, negative error code otherwise

bool **filemap_dirty_folio**(struct address_space *mapping, struct *folio* *folio)

Mark a folio dirty for filesystems which do not use buffer_heads.

Parameters

struct address_space *mapping
Address space this folio belongs to.

struct folio *folio
Folio to be marked as dirty.

Description

Filesystems which do not use buffer heads should call this function from their set_page_dirty address space operation. It ignores the contents of folio_get_private(), so if the filesystem marks individual blocks as dirty, the filesystem should handle that itself.

This is also sometimes used by filesystems which use buffer_heads when a single buffer is being dirtied: we want to set the folio dirty in that case, but not all the buffers. This is a "bottom-up" dirtying, whereas block_dirty_folio() is a "top-down" dirtying.

The caller must ensure this doesn't race with truncation. Most will simply hold the folio lock, but e.g. zap_pte_range() calls with the folio mapped and the pte lock held, which also locks out truncation.

bool **folio_redirty_for_writepage**(struct writeback_control *wbc, struct *folio* *folio)

Decline to write a dirty folio.

Parameters

struct writeback_control *wbc
The writeback control.

struct folio *folio
The folio.

Description

When a writepage implementation decides that it doesn't want to write **folio** for some reason, it should call this function, unlock **folio** and return 0.

Return

True if we redirtied the folio. False if someone else dirtied it first.

bool **folio_mark_dirty**(struct *folio* *folio)

Mark a folio as being modified.

Parameters

struct folio *folio
The folio.

Description

The folio may not be truncated while this function is running. Holding the folio lock is sufficient to prevent truncation, but some callers cannot acquire a sleeping lock. These callers instead hold the page table lock for a page table which contains at least one page in this folio. Truncation will block on the page table lock as it unmaps pages before removing the folio from its mapping.

Return

True if the folio was newly dirtied, false if it was already dirty.

void **folio_wait_writeback**(struct *folio* *folio)

Wait for a folio to finish writeback.

Parameters

struct folio *folio
The folio to wait for.

Description

If the folio is currently being written back to storage, wait for the I/O to complete.

Context

Sleeps. Must be called in process context and with no spinlocks held. Caller should hold a reference on the folio. If the folio is not locked, writeback may start again after writeback has finished.

int **folio_wait_writeback_killable**(struct *folio* *folio)

Wait for a folio to finish writeback.

Parameters

struct folio *folio
The folio to wait for.

Description

If the folio is currently being written back to storage, wait for the I/O to complete or a fatal signal to arrive.

Context

Sleeps. Must be called in process context and with no spinlocks held. Caller should hold a reference on the folio. If the folio is not locked, writeback may start again after writeback has finished.

Return

0 on success, -EINTR if we get a fatal signal while waiting.

void **folio_wait_stable**(struct *folio* *folio)

wait for writeback to finish, if necessary.

Parameters

struct folio *folio
The folio to wait on.

Description

This function determines if the given folio is related to a backing device that requires folio contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete.

Context

Sleeps. Must be called in process context and with no spinlocks held. Caller should hold a reference on the folio. If the folio is not locked, writeback may start again after writeback has finished.

Truncate

void **folio_invalidate**(struct *folio* *folio, size_t offset, size_t length)

Invalidate part or all of a folio.

Parameters

struct folio *folio
The folio which is affected.

size_t offset
start of the range to invalidate

size_t length
length of the range to invalidate

Description

folio_invalidate() is called when all or part of the folio has become invalidated by a truncate operation.

folio_invalidate() does not have to release all buffers, but it must ensure that no dirty buffer is left outside **offset** and that no I/O is underway against any of the blocks which are outside the truncation point. Because the caller is about to free (and possibly reuse) those blocks on-disk.

void **truncate_inode_pages_range**(struct address_space *mapping, loff_t lstart, loff_t lend)

truncate range of pages specified by start & end byte offsets

Parameters

struct address_space *mapping
mapping to truncate

loff_t lstart
offset from which to truncate

loff_t lend
offset to which to truncate (inclusive)

Description

Truncate the page cache, removing the pages that are between specified offsets (and zeroing out partial pages if lstart or lend + 1 is not page aligned).

Truncate takes two passes - the first pass is nonblocking. It will not block on page locks and it will not block on writeback. The second pass will wait. This is to prevent as much IO as possible in the affected region. The first pass will remove most pages, so the search cost of the second pass is low.

We pass down the cache-hot hint to the page freeing code. Even if the mapping is large, it is probably the case that the final pages are the most recently touched, and freeing happens in ascending file offset order.

Note that since ->invalidate_folio() accepts range to invalidate truncate_inode_pages_range is able to handle cases where lend + 1 is not page aligned properly.

void **truncate_inode_pages**(struct address_space *mapping, loff_t lstart)
truncate *all* the pages from an offset

Parameters

struct address_space *mapping
mapping to truncate

loff_t lstart
offset from which to truncate

Description

Called under (and serialised by) inode->i_rwsem and mapping->invalidate_lock.

Note

When this function returns, there can be a page in the process of deletion (inside __filemap_remove_folio()) in the specified range. Thus mapping->nrpages can be non-zero when this function returns even after truncation of the whole mapping.

void **truncate_inode_pages_final**(struct address_space *mapping)
truncate *all* pages before inode dies

Parameters

struct address_space *mapping
mapping to truncate

Description

Called under (and serialized by) inode->i_rwsem.

Filesystems have to use this in the .evict_inode path to inform the VM that this is the final truncate and the inode is going away.

unsigned long **invalidate_mapping_pages**(struct address_space *mapping, pgoff_t start, pgoff_t end)

Invalidate all clean, unlocked cache of one inode

Parameters

struct address_space *mapping
the address_space which holds the cache to invalidate

pgoff_t start
the offset 'from' which to invalidate

pgoff_t end
the offset 'to' which to invalidate (inclusive)

Description

This function removes pages that are clean, unmapped and unlocked, as well as shadow entries. It will not block on IO activity.

If you want to remove all the pages of one inode, regardless of their use and writeback state, use *truncate_inode_pages()*.

Return

The number of indices that had their contents invalidated

int **invalidate_inode_pages2_range**(struct address_space *mapping, pgoff_t start, pgoff_t end)

remove range of pages from an address_space

Parameters

struct address_space *mapping
the address_space

pgoff_t start
the page offset 'from' which to invalidate

pgoff_t end
the page offset 'to' which to invalidate (inclusive)

Description

Any pages which are found to be mapped into pagetables are unmapped prior to invalidation.

Return

-EBUSY if any pages could not be invalidated.

int **invalidate_inode_pages2**(struct address_space *mapping)

remove all pages from an address_space

Parameters

struct address_space *mapping
the address_space

Description

Any pages which are found to be mapped into pagetables are unmapped prior to invalidation.

Return

-EBUSY if any pages could not be invalidated.

void **truncate_pagecache**(struct *inode* *inode, loff_t newsize)
unmap and remove pagecache that has been truncated

Parameters

struct inode *inode
inode

loff_t newsize
new file size

Description

inode's new i_size must already be written before truncate_pagecache is called.

This function should typically be called before the filesystem releases resources associated with the freed range (eg. deallocates blocks). This way, pagecache will always stay logically coherent with on-disk format, and the filesystem would not have to deal with situations such as writepage being called for a page that has already had its underlying blocks deallocated.

void **truncate_setsize**(struct *inode* *inode, loff_t newsize)
update inode and pagecache for a new file size

Parameters

struct inode *inode
inode

loff_t newsize
new file size

Description

truncate_setsize updates i_size and performs pagecache truncation (if necessary) to **newsize**. It will be typically be called from the filesystem's setattr function when ATTR_SIZE is passed in.

Must be called with a lock serializing truncates and writes (generally i_rwsem but e.g. xfs uses a different lock) and before all filesystem specific block truncation has been performed.

void **pagecache_isize_extended**(struct *inode* *inode, loff_t from, loff_t to)
update pagecache after extension of i_size

Parameters

struct inode *inode
inode for which i_size was extended

loff_t from
original inode size

loff_t to
new inode size

Description

Handle extension of inode size either caused by extending truncate or by write starting after current i_size. We mark the page straddling current i_size RO so that page_mkwrite() is called on the nearest write access to the page. This way filesystem can be sure that page_mkwrite() is called on the page before user writes to the page via mmap after the i_size has been changed.

The function must be called after i_size is updated so that page fault coming after we unlock the page will already see the new i_size. The function must be called while we still hold i_rwsem - this not only makes sure i_size is stable but also that userspace cannot observe new i_size value before we are prepared to store mmap writes at new inode size.

void **truncate_pagecache_range**(struct *inode* *inode, loff_t lstart, loff_t lend)

unmap and remove pagecache that is hole-punched

Parameters

struct inode *inode
inode

loff_t lstart
offset of beginning of hole

loff_t lend
offset of last byte of hole

Description

This function should typically be called before the filesystem releases resources associated with the freed range (eg. deallocates blocks). This way, pagecache will always stay logically coherent with on-disk format, and the filesystem would not have to deal with situations such as writepage being called for a page that has already had its underlying blocks deallocated.

void **filemap_set_wb_err**(struct address_space *mapping, int err)

set a writeback error on an address_space

Parameters

struct address_space *mapping
mapping in which to set writeback error

int err
error to be set in mapping

Description

When writeback fails in some way, we must record that error so that userspace can be informed when fsync and the like are called. We endeavor to report errors on any file that was open at the time of the error. Some internal callers also need to know when writeback errors have occurred.

When a writeback error occurs, most filesystems will want to call filemap_set_wb_err to record the error in the mapping so that it will be automatically reported whenever fsync is called on the file.

int **filemap_check_wb_err**(struct address_space *mapping, errseq_t since)

has an error occurred since the mark was sampled?

Parameters

struct address_space *mapping
mapping to check for writeback errors

errseq_t since
previously-sampled errseq_t

Description

Grab the errseq_t value from the mapping, and see if it has changed “since” the given value was sampled.

If it has then report the latest error set, otherwise return 0.

errseq_t **filemap_sample_wb_err**(struct address_space *mapping)
sample the current errseq_t to test for later errors

Parameters

struct address_space *mapping
mapping to be sampled

Description

Writeback errors are always reported relative to a particular sample point in the past. This function provides those sample points.

errseq_t **file_sample_sb_err**(struct *file* *file)
sample the current errseq_t to test for later errors

Parameters

struct file *file
file pointer to be sampled

Description

Grab the most current superblock-level errseq_t value for the given struct file.

void **mapping_set_error**(struct address_space *mapping, int error)
record a writeback error in the address_space

Parameters

struct address_space *mapping
the mapping in which an error should be set

int error
the error to set in the mapping

Description

When writeback fails in some way, we must record that error so that userspace can be informed when fsync and the like are called. We endeavor to report errors on any file that was open at the time of the error. Some internal callers also need to know when writeback errors have occurred.

When a writeback error occurs, most filesystems will want to call mapping_set_error to record the error in the mapping so that it can be reported when the application calls fsync(2).

void **mapping_set_large_folios**(struct address_space *mapping)
Indicate the file supports large folios.

Parameters

struct address_space *mapping
The file.

Description

The filesystem should call this function in its inode constructor to indicate that the VFS can use large folios to cache the contents of the file.

Context

This should not be called while the inode is active as it is non-atomic.

struct address_space ***folio_file_mapping**(struct *folio* *folio)
Find the mapping this folio belongs to.

Parameters

struct folio *folio
The folio.

Description

For folios which are in the page cache, return the mapping that this page belongs to. Folios in the swap cache return the mapping of the swap file or swap device where the data is stored. This is different from the mapping returned by `folio_mapping()`. The only reason to use it is if, like NFS, you return 0 from ->activate_swapfile.

Do not call this for folios which aren't in the page cache or swap cache.

struct address_space ***folio_flush_mapping**(struct *folio* *folio)
Find the file mapping this folio belongs to.

Parameters

struct folio *folio
The folio.

Description

For folios which are in the page cache, return the mapping that this page belongs to. Anonymous folios return NULL, even if they're in the swap cache. Other kinds of folio also return NULL.

This is ONLY used by architecture cache flushing code. If you aren't writing cache flushing code, you want either `folio_mapping()` or `folio_file_mapping()`.

struct inode ***folio_inode**(struct *folio* *folio)
Get the host inode for this folio.

Parameters

struct folio *folio
The folio.

Description

For folios which are in the page cache, return the inode that this folio belongs to.

Do not call this for folios which aren't in the page cache.

void **folio_attach_private**(struct *folio* *folio, void *data)
Attach private data to a folio.

Parameters

struct folio *folio
Folio to attach data to.

void *data
Data to attach to folio.

Description

Attaching private data to a folio increments the page's reference count. The data must be detached before the folio will be freed.

void ***folio_change_private**(struct *folio* *folio, void *data)
Change private data on a folio.

Parameters

struct folio *folio
Folio to change the data on.

void *data
Data to set on the folio.

Description

Change the private data attached to a folio and return the old data. The page must previously have had data attached and the data must be detached before the folio will be freed.

Return

Data that was previously attached to the folio.

void ***folio_detach_private**(struct *folio* *folio)
Detach private data from a folio.

Parameters

struct folio *folio
Folio to detach data from.

Description

Removes the data that was previously attached to the folio and decrements the refcount on the page.

Return

Data that was attached to the folio.

type **fgf_t**
Flags for getting folios from the page cache.

Description

Most users of the page cache will not need to use these flags; there are convenience functions such as `filemap_get_folio()` and `filemap_lock_folio()`. For users which need more control over exactly what is done with the folios, these flags to `__filemap_get_folio()` are available.

- `FGP_ACCESSED` - The folio will be marked accessed.
- `FGP_LOCK` - The folio is returned locked.

- FGP_CREAT - If no folio is present then a new folio is allocated, added to the page cache and the VM's LRU list. The folio is returned locked.
- FGP_FOR_MMAP - The caller wants to do its own locking dance if the folio is already in cache. If the folio was allocated, unlock it before returning so the caller can do the same dance.
- FGP_WRITE - The folio will be written to by the caller.
- FGP_NOFS - __GFP_FS will get cleared in gfp.
- FGP_NOWAIT - Don't block on the folio lock.
- FGP_STABLE - Wait for the folio to be stable (finished writeback)
- FGP_WRITEBEGIN - The flags to use in a filesystem write_begin() implementation.

fgf_t **fgf_set_order**(size_t size)

Encode a length in the fgf_t flags.

Parameters

size_t size
The suggested size of the folio to create.

Description

The caller of *__filemap_get_folio()* can use this to suggest a preferred size for the folio that is created. If there is already a folio at the index, it will be returned, no matter what its size. If a folio is freshly created, it may be of a different size than requested due to alignment constraints, memory pressure, or the presence of other folios at nearby indices.

struct *folio* ***filemap_get_folio**(struct address_space *mapping, pgoff_t index)

Find and get a folio.

Parameters

struct address_space *mapping
The address_space to search.

pgoff_t index
The page index.

Description

Looks up the page cache entry at **mapping** & **index**. If a folio is present, it is returned with an increased refcount.

Return

A folio or ERR_PTR(-ENOENT) if there is no folio in the cache for this index. Will not return a shadow, swap or DAX entry.

struct *folio* ***filemap_lock_folio**(struct address_space *mapping, pgoff_t index)

Find and lock a folio.

Parameters

struct address_space *mapping
The address_space to search.

pgoff_t index
The page index.

Description

Looks up the page cache entry at **mapping** & **index**. If a folio is present, it is returned locked with an increased refcount.

Context

May sleep.

Return

A folio or ERR_PTR(-ENOENT) if there is no folio in the cache for this index. Will not return a shadow, swap or DAX entry.

struct *folio* ***filemap_grab_folio**(struct address_space *mapping, pgoff_t index)

grab a folio from the page cache

Parameters

struct address_space *mapping
The address space to search

pgoff_t index
The page index

Description

Looks up the page cache entry at **mapping** & **index**. If no folio is found, a new folio is created. The folio is locked, marked as accessed, and returned.

Return

A found or created folio. ERR_PTR(-ENOMEM) if no folio is found and failed to create a folio.

struct page ***find_get_page**(struct address_space *mapping, pgoff_t offset)

find and get a page reference

Parameters

struct address_space *mapping
the address_space to search

pgoff_t offset
the page index

Description

Looks up the page cache slot at **mapping** & **offset**. If there is a page cache page, it is returned with an increased refcount.

Otherwise, `NULL` is returned.

struct page ***find_lock_page**(struct address_space *mapping, pgoff_t index)

locate, pin and lock a pagecache page

Parameters

struct address_space *mapping
the address_space to search

pgoff_t index
the page index

Description

Looks up the page cache entry at **mapping** & **index**. If there is a page cache page, it is returned locked and with an increased refcount.

Context

May sleep.

Return

A struct page or `NULL` if there is no page in the cache for this index.

struct page ***`find_or_create_page`**(struct address_space *mapping, pgoff_t index, gfp_t gfp_mask)

locate or add a pagecache page

Parameters

`struct address_space *mapping`
the page's address_space

`pgoff_t index`
the page's index into the mapping

`gfp_t gfp_mask`
page allocation mode

Description

Looks up the page cache slot at **mapping** & **offset**. If there is a page cache page, it is returned locked and with an increased refcount.

If the page is not present, a new page is allocated using **gfp_mask** and added to the page cache and the VM's LRU list. The page is returned locked and with an increased refcount.

On memory exhaustion, `NULL` is returned.

`find_or_create_page()` may sleep, even if **gfp_flags** specifies an atomic allocation!

struct page ***`grab_cache_page_nowait`**(struct address_space *mapping, pgoff_t index)

returns locked page at given index in given cache

Parameters

`struct address_space *mapping`
target address_space

`pgoff_t index`
the page index

Description

Same as grab_cache_page(), but do not wait if the page is unavailable. This is intended for speculative data generators, where the data can be regenerated if the page couldn't be grabbed. This routine should be safe to call while holding the lock for another page.

Clear __GFP_FS when allocating the page to avoid recursion into the fs and deadlock against the caller's locked page.

pgoff_t **folio_index**(struct *folio* *folio)

File index of a folio.

Parameters

struct folio *folio
The folio.

Description

For a folio which is either in the page cache or the swap cache, return its index within the address_space it belongs to. If you know the page is definitely in the page cache, you can look at the folio's index directly.

Return

The index (offset in units of pages) of a folio in its file.

pgoff_t **folio_next_index**(struct *folio* *folio)

Get the index of the next folio.

Parameters

struct folio *folio
The current folio.

Return

The index of the folio which follows this folio in the file.

struct page ***folio_file_page**(struct *folio* *folio, pgoff_t index)

The page for a particular index.

Parameters

struct folio *folio
The folio which contains this index.

pgoff_t index
The index we want to look up.

Description

Sometimes after looking up a folio in the page cache, we need to obtain the specific page for an index (eg a page fault).

Return

The page containing the file data for this index.

bool **folio_contains**(struct *folio* *folio, pgoff_t index)

Does this folio contain this index?

Parameters

struct folio *folio
The folio.

pgoff_t index
The page index within the file.

Context

The caller should have the page locked in order to prevent (eg) shmem from moving the page between the page cache and swap cache and changing its index in the middle of the operation.

Return

true or false.

loff_t **folio_pos**(struct *folio* *folio)

Returns the byte position of this folio in its file.

Parameters

struct folio *folio
The folio.

loff_t **folio_file_pos**(struct *folio* *folio)

Returns the byte position of this folio in its file.

Parameters

struct folio *folio
The folio.

Description

This differs from `folio_pos()` for folios which belong to a swap file. NFS is the only filesystem today which needs to use `folio_file_pos()`.

bool **folio_trylock**(struct *folio* *folio)

Attempt to lock a folio.

Parameters

struct folio *folio
The folio to attempt to lock.

Description

Sometimes it is undesirable to wait for a folio to be unlocked (eg when the locks are being taken in the wrong order, or if making progress through a batch of folios is more important than processing them in order). Usually `folio_lock()` is the correct function to call.

Context

Any context.

Return

Whether the lock was successfully acquired.

void **folio_lock**(struct *folio* *folio)

Lock this folio.

Parameters

struct folio *folio
The folio to lock.

Description

The folio lock protects against many things, probably more than it should. It is primarily held while a folio is being brought uptodate, either from its backing file or from swap. It is also held while a folio is being truncated from its address_space, so holding the lock is sufficient to keep folio->mapping stable.

The folio lock is also held while write() is modifying the page to provide POSIX atomicity guarantees (as long as the write does not cross a page boundary). Other modifications to the data in the folio do not hold the folio lock and can race with writes, eg DMA and stores to mapped pages.

Context

May sleep. If you need to acquire the locks of two or more folios, they must be in order of ascending index, if they are in the same address_space. If they are in different address_spaces, acquire the lock of the folio which belongs to the address_space which has the lowest address in memory first.

void **lock_page**(struct *page* *page)

Lock the folio containing this page.

Parameters

struct page *page
The page to lock.

Description

See *folio_lock()* for a description of what the lock protects. This is a legacy function and new code should probably use *folio_lock()* instead.

Context

May sleep. Pages in the same folio share a lock, so do not attempt to lock two pages which share a folio.

int **folio_lock_killable**(struct *folio* *folio)

Lock this folio, interruptible by a fatal signal.

Parameters

struct folio *folio
The folio to lock.

Description

Attempts to lock the folio, like *folio_lock()*, except that the sleep to acquire the lock is interruptible by a fatal signal.

Context

May sleep; see *folio_lock()*.

Return

0 if the lock was acquired; -EINTR if a fatal signal was received.

bool **filemap_range_needs_writeback**(struct address_space *mapping, loff_t start_byte, loff_t end_byte)

check if range potentially needs writeback

Parameters

struct address_space *mapping
address space within which to check

loff_t start_byte
offset in bytes where the range starts

loff_t end_byte
offset in bytes where the range ends (inclusive)

Description

Find at least one page in the range supplied, usually used to check if direct writing in this range will trigger a writeback. Used by O_DIRECT read/write with IOCB_NOWAIT, to see if the caller needs to do filemap_write_and_wait_range() before proceeding.

Return

`true` if the caller should do filemap_write_and_wait_range() before doing O_DIRECT to a page in this range, `false` otherwise.

struct **readahead_control**

Describes a readahead request.

Definition:

```
struct readahead_control {
    struct file *file;
    struct address_space *mapping;
    struct file_ra_state *ra;
};
```

Members

file
The file, used primarily by network filesystems for authentication. May be NULL if invoked internally by the filesystem.

mapping
Readahead this filesystem object.

ra
File readahead state. May be NULL.

Description

A readahead request is for consecutive pages. Filesystems which implement the ->readahead method should call *readahead_page()* or *readahead_page_batch()* in a loop and attempt to start I/O against each page in the request.

Most of the fields in this struct are private and should be accessed by the functions below.

void **page_cache_sync_readahead**(struct address_space *mapping, struct file_ra_state *ra, struct *file* *file, pgoff_t index, unsigned long req_count)

generic file readahead

Parameters

struct address_space *mapping
address_space which holds the pagecache and I/O vectors

struct file_ra_state *ra
file_ra_state which holds the readahead state

struct file *file
Used by the filesystem for authentication.

pgoff_t index
Index of first page to be read.

unsigned long req_count
Total number of pages being read by the caller.

Description

page_cache_sync_readahead() should be called when a cache miss happened: it will submit the read. The readahead logic may decide to piggyback more pages onto the read request if access patterns suggest it will improve performance.

void **page_cache_async_readahead**(struct address_space *mapping, struct file_ra_state *ra, struct *file* *file, struct *folio* *folio, pgoff_t index, unsigned long req_count)

file readahead for marked pages

Parameters

struct address_space *mapping
address_space which holds the pagecache and I/O vectors

struct file_ra_state *ra
file_ra_state which holds the readahead state

struct file *file
Used by the filesystem for authentication.

struct folio *folio
The folio at **index** which triggered the readahead call.

pgoff_t index
Index of first page to be read.

unsigned long req_count
Total number of pages being read by the caller.

Description

page_cache_async_readahead() should be called when a page is used which is marked as PageReadahead; this is a marker to suggest that the application has used up enough of the readahead window that we should start pulling in more pages.

struct page ***readahead_page**(struct *readahead_control* *ractl)

Get the next page to read.

Parameters

struct readahead_control *ractl
The current readahead request.

Context

The page is locked and has an elevated refcount. The caller should decreases the refcount once the page has been submitted for I/O and unlock the page once all I/O to that page has completed.

Return

A pointer to the next page, or `NULL` if we are done.

struct *folio* ***readahead_folio**(struct *readahead_control* *ractl)

Get the next folio to read.

Parameters

struct readahead_control *ractl
The current readahead request.

Context

The folio is locked. The caller should unlock the folio once all I/O to that folio has completed.

Return

A pointer to the next folio, or `NULL` if we are done.

readahead_page_batch

`readahead_page_batch (rac, array)`

Get a batch of pages to read.

Parameters

rac
The current readahead request.

array
An array of pointers to struct page.

Context

The pages are locked and have an elevated refcount. The caller should decreases the refcount once the page has been submitted for I/O and unlock the page once all I/O to that page has completed.

Return

The number of pages placed in the array. 0 indicates the request is complete.

loff_t **readahead_pos**(struct *readahead_control* *rac)

The byte offset into the file of this readahead request.

Parameters

struct readahead_control *rac
The readahead request.

size_t **readahead_length**(struct *readahead_control* *rac)

The number of bytes in this readahead request.

Parameters

struct readahead_control *rac
The readahead request.

pgoff_t **readahead_index**(struct *readahead_control* *rac)

The index of the first page in this readahead request.

Parameters

struct readahead_control *rac
The readahead request.

unsigned int **readahead_count**(struct *readahead_control* *rac)

The number of pages in this readahead request.

Parameters

struct readahead_control *rac
The readahead request.

size_t **readahead_batch_length**(struct *readahead_control* *rac)

The number of bytes in the current batch.

Parameters

struct readahead_control *rac
The readahead request.

ssize_t **folio_mkwrite_check_truncate**(struct *folio* *folio, struct *inode* *inode)

check if folio was truncated

Parameters

struct folio *folio
the folio to check

struct inode *inode
the inode to check the folio against

Return

the number of bytes in the folio up to EOF, or -EFAULT if the folio was truncated.

int **page_mkwrite_check_truncate**(struct *page* *page, struct *inode* *inode)

check if page was truncated

Parameters

struct page *page
the page to check

struct inode *inode
the inode to check the page against

Description

Returns the number of bytes in the page up to EOF, or -EFAULT if the page was truncated.

unsigned int **i_blocks_per_folio**(struct *inode* *inode, struct *folio* *folio)

How many blocks fit in this folio.

Parameters

struct inode *inode
The inode which contains the blocks.

struct folio *folio
The folio.

Description

If the block size is larger than the size of this folio, return zero.

Context

The caller should hold a refcount on the folio to prevent it from being split.

Return

The number of filesystem blocks covered by this folio.

6.7.6 Memory pools

void **mempool_exit**(mempool_t *pool)
exit a mempool initialized with *mempool_init()*

Parameters

mempool_t *pool
pointer to the memory pool which was initialized with *mempool_init()*.

Description

Free all reserved elements in **pool** and **pool** itself. This function only sleeps if the free_fn() function sleeps.

May be called on a zeroed but uninitialized mempool (i.e. allocated with kzalloc()).

void **mempool_destroy**(mempool_t *pool)
deallocate a memory pool

Parameters

mempool_t *pool
pointer to the memory pool which was allocated via *mempool_create()*.

Description

Free all reserved elements in **pool** and **pool** itself. This function only sleeps if the free_fn() function sleeps.

int **mempool_init**(mempool_t *pool, int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data)
initialize a memory pool

Parameters

mempool_t *pool
pointer to the memory pool that should be initialized

int min_nr
the minimum number of elements guaranteed to be allocated for this pool.

mempool_alloc_t *alloc_fn
user-defined element-allocation function.

mempool_free_t *free_fn
user-defined element-freeing function.

void *pool_data
optional private data available to the user-defined functions.

Description

Like *mempool_create()*, but initializes the pool in (i.e. embedded in another structure).

Return

0 on success, negative error code otherwise.

mempool_t ***mempool_create**(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data)

create a memory pool

Parameters

int min_nr
the minimum number of elements guaranteed to be allocated for this pool.

mempool_alloc_t *alloc_fn
user-defined element-allocation function.

mempool_free_t *free_fn
user-defined element-freeing function.

void *pool_data
optional private data available to the user-defined functions.

Description

this function creates and allocates a guaranteed size, preallocated memory pool. The pool can be used from the *mempool_alloc()* and *mempool_free()* functions. This function might sleep. Both the alloc_fn() and the free_fn() functions might sleep - as long as the *mempool_alloc()* function is not called from IRQ contexts.

Return

pointer to the created memory pool object or NULL on error.

int **mempool_resize**(mempool_t *pool, int new_min_nr)

resize an existing memory pool

Parameters

mempool_t *pool
pointer to the memory pool which was allocated via *mempool_create()*.

int new_min_nr
the new minimum number of elements guaranteed to be allocated for this pool.

Description

This function shrinks/grows the pool. In the case of growing, it cannot be guaranteed that the pool will be grown to the new size immediately, but new *mempool_free()* calls will refill it. This function may sleep.

Note, the caller must guarantee that no mempool_destroy is called while this function is running. *mempool_alloc()* & *mempool_free()* might be called (eg. from IRQ contexts) while this function executes.

Return

`0` on success, negative error code otherwise.

void ***mempool_alloc**(mempool_t *pool, gfp_t gfp_mask)

allocate an element from a specific memory pool

Parameters

mempool_t *pool
pointer to the memory pool which was allocated via *mempool_create()*.

gfp_t gfp_mask
the usual allocation bitmask.

Description

this function only sleeps if the alloc_fn() function sleeps or returns NULL. Note that due to preallocation, this function *never* fails when called from process contexts. (it might fail if called from an IRQ context.)

Note

using __GFP_ZERO is not supported.

Return

pointer to the allocated element or `NULL` on error.

void **mempool_free**(void *element, mempool_t *pool)

return an element to the pool.

Parameters

void *element
pool element pointer.

mempool_t *pool
pointer to the memory pool which was allocated via *mempool_create()*.

Description

this function only sleeps if the free_fn() function sleeps.

6.7.7 DMA pools

struct dma_pool ***dma_pool_create**(const char *name, struct device *dev, size_t size, size_t align, size_t boundary)

Creates a pool of consistent memory blocks, for dma.

Parameters

const char *name
name of pool, for diagnostics

struct device *dev
device that will be doing the DMA

size_t size
size of the blocks in this pool.

size_t align
alignment requirement for blocks; must be a power of two

size_t boundary
returned blocks won't cross this power of two boundary

Context

not in_interrupt()

Description

Given one of these pools, *dma_pool_alloc()* may be used to allocate memory. Such memory will all have "consistent" DMA mappings, accessible by the device and its driver without using cache flushing primitives. The actual size of blocks allocated may be larger than requested because of alignment.

If **boundary** is nonzero, objects returned from *dma_pool_alloc()* won't cross that size boundary. This is useful for devices which have addressing restrictions on individual DMA transfers, such as not crossing boundaries of 4KBytes.

Return

a dma allocation pool with the requested characteristics, or `NULL` if one can't be created.

void **dma_pool_destroy**(struct dma_pool *pool)
destroys a pool of dma memory blocks.

Parameters

struct dma_pool *pool
dma pool that will be destroyed

Context

!in_interrupt()

Description

Caller guarantees that no more memory from the pool is in use, and that nothing will try to use the pool after this call.

void ***dma_pool_alloc**(struct dma_pool *pool, gfp_t mem_flags, dma_addr_t *handle)
get a block of consistent memory

Parameters

struct dma_pool *pool
dma pool that will produce the block

gfp_t mem_flags
GFP_* bitmask

dma_addr_t *handle
pointer to dma address of block

Return

the kernel virtual address of a currently unused block, and reports its dma address through the handle. If such a memory block can't be allocated, NULL is returned.

void **dma_pool_free**(struct dma_pool *pool, void *vaddr, dma_addr_t dma)

put block back into dma pool

Parameters

struct dma_pool *pool
the dma pool holding the block

void *vaddr
virtual address of block

dma_addr_t dma
dma address of block

Description

Caller promises neither device nor driver will again touch this block unless it is first re-allocated.

struct dma_pool ***dmam_pool_create**(const char *name, struct device *dev, size_t size, size_t align, size_t allocation)

Managed *dma_pool_create()*

Parameters

const char *name
name of pool, for diagnostics

struct device *dev
device that will be doing the DMA

size_t size
size of the blocks in this pool.

size_t align
alignment requirement for blocks; must be a power of two

size_t allocation
returned blocks won't cross this boundary (or zero)

Description

Managed *dma_pool_create()*. DMA pool created with this function is automatically destroyed on driver detach.

Return

a managed dma allocation pool with the requested characteristics, or NULL if one can't be created.

void **dmam_pool_destroy**(struct dma_pool *pool)

Managed *dma_pool_destroy()*

Parameters

struct dma_pool *pool
dma pool that will be destroyed

Description

Managed *dma_pool_destroy()*.

6.7.8 More Memory Management Functions

void **zap_vma_ptes**(struct vm_area_struct *vma, unsigned long address, unsigned long size)
remove ptes mapping the vma

Parameters

struct vm_area_struct *vma
vm_area_struct holding ptes to be zapped

unsigned long address
starting address of pages to zap

unsigned long size
number of bytes to zap

Description

This function only unmaps ptes assigned to VM_PFNMAP vmas.

The entire address range must be fully contained within the vma.

int **vm_insert_pages**(struct vm_area_struct *vma, unsigned long addr, struct page **pages, unsigned long *num)
insert multiple pages into user vma, batching the pmd lock.

Parameters

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target start user address of these pages

struct page **pages
source kernel pages

unsigned long *num
in: number of pages to map. out: number of pages that were *not* mapped. (0 means all pages were successfully mapped).

Description

Preferred over *vm_insert_page()* when inserting multiple pages.

In case of error, we may have mapped a subset of the provided pages. It is the caller's responsibility to account for this case.

The same restrictions apply as in *vm_insert_page()*.

int **vm_insert_page**(struct vm_area_struct *vma, unsigned long addr, struct *page* *page)
insert single page into user vma

Parameters

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target user address of this page

struct page *page
source kernel page

Description

This allows drivers to insert individual pages they've allocated into a user vma.

The page has to be a nice clean _individual_ kernel allocation. If you allocate a compound page, you need to have marked it as such (__GFP_COMP), or manually just split the page up yourself (see split_page()).

NOTE! Traditionally this was done with "*remap_pfn_range()*" which took an arbitrary page protection parameter. This doesn't allow that. Your vma protection will have to be set up correctly, which means that if you want a shared writable mapping, you'd better ask for a shared writable mapping!

The page does not need to be reserved.

Usually this function is called from f_op->mmap() handler under mm->mmap_lock write-lock, so it can change vma->vm_flags. Caller must set VM_MIXEDMAP on vma if it wants to call this function from other places, for example from page-fault handler.

Return

`0` on success, negative error code otherwise.

int **vm_map_pages**(struct vm_area_struct *vma, struct page **pages, unsigned long num)
maps range of kernel pages starts with non zero offset

Parameters

struct vm_area_struct *vma
user vma to map to

struct page **pages
pointer to array of source kernel pages

unsigned long num
number of pages in page array

Description

Maps an object consisting of **num** pages, catering for the user's requested vm_pgoff

If we fail to insert any page into the vma, the function will return immediately leaving any previously inserted pages present. Callers from the mmap handler may immediately return the error as their caller will destroy the vma, removing any successfully inserted pages. Other callers should make their own arrangements for calling unmap_region().

Context

Process context. Called by mmap handlers.

Return

0 on success and error code otherwise.

int **vm_map_pages_zero**(struct vm_area_struct *vma, struct page **pages, unsigned long num)

map range of kernel pages starts with zero offset

Parameters

struct vm_area_struct *vma
user vma to map to

struct page **pages
pointer to array of source kernel pages

unsigned long num
number of pages in page array

Description

Similar to *vm_map_pages()*, except that it explicitly sets the offset to 0. This function is intended for the drivers that did not consider vm_pgoff.

Context

Process context. Called by mmap handlers.

Return

0 on success and error code otherwise.

vm_fault_t **vmf_insert_pfn_prot**(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, pgprot_t pgprot)

insert single pfn into user vma with specified pgprot

Parameters

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target user address of this page

unsigned long pfn
source kernel pfn

pgprot_t pgprot
pgprot flags for the inserted page

Description

This is exactly like *vmf_insert_pfn()*, except that it allows drivers to override pgprot on a per-page basis.

This only makes sense for IO mappings, and it makes no sense for COW mappings. In general, using multiple vmas is preferable; vmf_insert_pfn_prot should only be used if using multiple VMAs is impractical.

pgprot typically only differs from **vma->vm_page_prot** when drivers set caching- and encryption bits different than those of **vma->vm_page_prot**, because the caching- or encryption mode may not be known at mmap() time.

This is ok as long as **vma->vm_page_prot** is not used by the core vm to set caching and encryption bits for those vmas (except for COW pages). This is ensured by core vm only modifying these page table entries using functions that don't touch caching- or encryption bits, using pte_modify() if needed. (See for example mprotect()).

Also when new page-table entries are created, this is only done using the fault() callback, and never using the value of vma->vm_page_prot, except for page-table entries that point to anonymous pages as the result of COW.

Context

Process context. May allocate using `GFP_KERNEL`.

Return

vm_fault_t value.

vm_fault_t **vmf_insert_pfn**(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn)

insert single pfn into user vma

Parameters

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target user address of this page

unsigned long pfn
source kernel pfn

Description

Similar to vm_insert_page, this allows drivers to insert individual pages they've allocated into a user vma. Same comments apply.

This function should only be called from a vm_ops->fault handler, and in that case the handler should return the result of this function.

vma cannot be a COW mapping.

As this is called only for pages that do not currently exist, we do not need to flush old virtual caches or the TLB.

Context

Process context. May allocate using `GFP_KERNEL`.

Return

vm_fault_t value.

int **remap_pfn_range**(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, unsigned long size, pgprot_t prot)

remap kernel memory to userspace

Parameters

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target page aligned user address to start at

unsigned long pfn
page frame number of kernel physical memory address

unsigned long size
size of mapping area

pgprot_t prot
page protection flags for this mapping

Note

this is only safe if the mm semaphore is held when called.

Return

0 on success, negative error code otherwise.

int **vm_iomap_memory**(struct vm_area_struct *vma, phys_addr_t start, unsigned long len)
remap memory to userspace

Parameters

struct vm_area_struct *vma
user vma to map to

phys_addr_t start
start of the physical memory to be mapped

unsigned long len
size of area

Description

This is a simplified io_remap_pfn_range() for common driver use. The driver just needs to give us the physical memory range to be mapped, we'll figure out the rest from the vma information.

NOTE! Some drivers might want to tweak vma->vm_page_prot first to get whatever write-combining details or similar.

Return

0 on success, negative error code otherwise.

void **unmap_mapping_pages**(struct address_space *mapping, pgoff_t start, pgoff_t nr, bool even_cows)

Unmap pages from processes.

Parameters

struct address_space *mapping
The address space containing pages to be unmapped.

pgoff_t start
Index of first page to be unmapped.

pgoff_t nr
Number of pages to be unmapped. 0 to unmap to end of file.

bool even_cows
Whether to unmap even private COWed pages.

Description

Unmap the pages in this address space from any userspace process which has them mmaped. Generally, you want to remove COWed pages as well when a file is being truncated, but not when invalidating pages from the page cache.

void **unmap_mapping_range**(struct address_space *mapping, loff_t const holebegin, loff_t const holelen, int even_cows)

unmap the portion of all mmaps in the specified address_space corresponding to the specified byte range in the underlying file.

Parameters

struct address_space *mapping
the address space containing mmaps to be unmapped.

loff_t const holebegin
byte in first page to unmap, relative to the start of the underlying file. This will be rounded down to a PAGE_SIZE boundary. Note that this is different from *truncate_pagecache()*, which must keep the partial page. In contrast, we must get rid of partial pages.

loff_t const holelen
size of prospective hole in bytes. This will be rounded up to a PAGE_SIZE boundary. A holelen of zero truncates to the end of the file.

int even_cows
1 when truncating a file, unmap even private COWed pages; but 0 when invalidating pagecache, don't throw away private data.

int **follow_pte**(struct mm_struct *mm, unsigned long address, pte_t **ptepp, spinlock_t **ptlp)

look up PTE at a user virtual address

Parameters

struct mm_struct *mm
the mm_struct of the target address space

unsigned long address
user virtual address

pte_t **ptepp
location to store found PTE

spinlock_t **ptlp
location to store the lock for the PTE

Description

On a successful return, the pointer to the PTE is stored in **ptepp**; the corresponding lock is taken and its location is stored in **ptlp**. The contents of the PTE are only stable until **ptlp** is released; any further use, if any, must be protected against invalidation with MMU notifiers.

Only IO mappings and raw PFN mappings are allowed. The mmap semaphore should be taken for read.

KVM uses this function. While it is arguably less bad than `follow_pfn`, it is not a good general-purpose API.

Return

zero on success, -ve otherwise.

int **follow_pfn**(struct vm_area_struct *vma, unsigned long address, unsigned long *pfn)
look up PFN at a user virtual address

Parameters

struct vm_area_struct *vma
memory mapping

unsigned long address
user virtual address

unsigned long *pfn
location to store found PFN

Description

Only IO mappings and raw PFN mappings are allowed.

This function does not allow the caller to read the permissions of the PTE. Do not use it.

Return

zero and the pfn at **pfn** on success, -ve otherwise.

int **generic_access_phys**(struct vm_area_struct *vma, unsigned long addr, void *buf, int len, int write)

generic implementation for iomem mmap access

Parameters

struct vm_area_struct *vma
the vma to access

unsigned long addr
userspace address, not relative offset within **vma**

void *buf
buffer to read/write

int len
length of transfer

int write
set to FOLL_WRITE when writing, otherwise reading

Description

This is a generic implementation for `vm_operations_struct.access` for an iomem mapping. This callback is used by access_process_vm() when the **vma** is not page based.

unsigned long **get_pfnblock_flags_mask**(const struct *page* *page, unsigned long pfn, unsigned long mask)

Return the requested group of flags for the pageblock_nr_pages block of pages

Parameters

const struct page *page
The page within the block of interest

unsigned long pfn
: The target page frame number

unsigned long mask
: mask of bits that the caller is interested in

Return

pageblock_bits flags

void **set_pfnblock_flags_mask**(struct *page* *page, unsigned long flags, unsigned long pfn, unsigned long mask)

Set the requested group of flags for a pageblock_nr_pages block of pages

Parameters

struct page *page
: The page within the block of interest

unsigned long flags
: The flags to set

unsigned long pfn
: The target page frame number

unsigned long mask
: mask of bits that the caller is interested in

int **split_free_page**(struct page *free_page, unsigned int order, unsigned long split_pfn_offset)

- split a free page at split_pfn_offset

Parameters

struct page *free_page
: the original free page

unsigned int order
: the order of the page

unsigned long split_pfn_offset
: split offset within the page

Description

Return -ENOENT if the free page is changed, otherwise 0

It is used when the free page crosses two pageblocks with different migratetypes at split_pfn_offset within the page. The split free page will be put into separate migratetype lists afterwards. Otherwise, the function achieves nothing.

void **__putback_isolated_page**(struct *page* *page, unsigned int order, int mt)

Return a now-isolated page back where we got it

Parameters

struct page *page
: Page that was isolated

unsigned int order
: Order of the isolated page

int mt
The page's pageblock's migratetype

Description

This function is meant to return a page pulled from the free lists via __isolate_free_page back to the free lists they were pulled from.

void **__free_pages**(struct *page* *page, unsigned int order)
Free pages allocated with *alloc_pages()*.

Parameters

struct page *page
The page pointer returned from *alloc_pages()*.

unsigned int order
The order of the allocation.

Description

This function can free multi-page allocations that are not compound pages. It does not check that the **order** passed in matches that of the allocation, so it is easy to leak memory. Freeing more memory than was allocated will probably emit a warning.

If the last reference to this page is speculative, it will be released by put_page() which only frees the first page of a non-compound allocation. To prevent the remaining pages from being leaked, we free the subsequent pages here. If you want to use the page's reference count to decide when to free the allocation, you should allocate a compound page, and use put_page() instead of *__free_pages()*.

Context

May be called in interrupt context or while holding a normal spinlock, but not in NMI context or while holding a raw spinlock.

void ***alloc_pages_exact**(size_t size, gfp_t gfp_mask)
allocate an exact number physically-contiguous pages.

Parameters

size_t size
the number of bytes to allocate

gfp_t gfp_mask
GFP flags for the allocation, must not contain __GFP_COMP

Description

This function is similar to *alloc_pages()*, except that it allocates the minimum number of pages to satisfy the request. *alloc_pages()* can only allocate memory in power-of-two pages.

This function is also limited by MAX_ORDER.

Memory allocated by this function must be released by *free_pages_exact()*.

Return

pointer to the allocated area or `NULL` in case of error.

void *`alloc_pages_exact_nid`(int nid, size_t size, gfp_t gfp_mask)
allocate an exact number of physically-contiguous pages on a node.

Parameters

int nid
the preferred node ID where memory should be allocated

size_t size
the number of bytes to allocate

gfp_t gfp_mask
GFP flags for the allocation, must not contain __GFP_COMP

Description

Like *alloc_pages_exact()*, but try to allocate on node nid first before falling back.

Return

pointer to the allocated area or `NULL` in case of error.

void `free_pages_exact`(void *virt, size_t size)
release memory allocated via *alloc_pages_exact()*

Parameters

void *virt
the value returned by alloc_pages_exact.

size_t size
size of allocation, same value as passed to *alloc_pages_exact()*.

Description

Release the memory allocated by a previous call to alloc_pages_exact.

unsigned long `nr_free_zone_pages`(int offset)
count number of pages beyond high watermark

Parameters

int offset
The zone index of the highest zone

Description

nr_free_zone_pages() counts the number of pages which are beyond the high watermark within all zones at or below a given zone index. For each zone, the number of pages is calculated as:

nr_free_zone_pages = managed_pages - high_pages

Return

number of pages beyond high watermark.

unsigned long `nr_free_buffer_pages`(void)
count number of pages beyond high watermark

Parameters

void
no arguments

Description

nr_free_buffer_pages() counts the number of pages which are beyond the high watermark within ZONE_DMA and ZONE_NORMAL.

Return

number of pages beyond high watermark within ZONE_DMA and ZONE_NORMAL.

int **find_next_best_node**(int node, nodemask_t *used_node_mask)
find the next node that should appear in a given node's fallback list

Parameters

int node
node whose fallback list we're appending

nodemask_t *used_node_mask
nodemask_t of already used nodes

Description

We use a number of factors to determine which is the next node that should appear on a given node's fallback list. The node should not have appeared already in **node**'s fallback list, and it should be the next closest node according to the distance array (which contains arbitrary distance values from each node to each node in the system), and should also prefer nodes with no CPUs, since presumably they'll have very little allocation pressure on them otherwise.

Return

node id of the found node or `NUMA_NO_NODE` if no node is found.

void **setup_per_zone_wmarks**(void)
called when min_free_kbytes changes or when memory is hot-{added|removed}

Parameters

void
no arguments

Description

Ensures that the watermark[min,low,high] values for each zone are set correctly with respect to min_free_kbytes.

int **alloc_contig_range**(unsigned long start, unsigned long end, unsigned migratetype, gfp_t gfp_mask)

- tries to allocate given range of pages

Parameters

unsigned long start
start PFN to allocate

unsigned long end
one-past-the-last PFN to allocate

unsigned migratetype
migratetype of the underlying pageblocks (either #MIGRATE_MOVABLE or #MIGRATE_CMA). All pageblocks in range must have the same migratetype and it must be either of the two.

gfp_t gfp_mask
GFP mask to use during compaction

Description

The PFN range does not have to be pageblock aligned. The PFN range must belong to a single zone.

The first thing this routine does is attempt to MIGRATE_ISOLATE all pageblocks in the range. Once isolated, the pageblocks should not be modified by others.

Return

zero on success or negative error code. On success all pages which PFN is in [start, end) are allocated for the caller and need to be freed with free_contig_range().

struct page ***alloc_contig_pages**(unsigned long nr_pages, gfp_t gfp_mask, int nid, nodemask_t *nodemask)

- tries to find and allocate contiguous range of pages

Parameters

unsigned long nr_pages
Number of contiguous pages to allocate

gfp_t gfp_mask
GFP mask to limit search and used during compaction

int nid
Target node

nodemask_t *nodemask
Mask for other possible nodes

Description

This routine is a wrapper around *alloc_contig_range()*. It scans over zones on an applicable zonelist to find a contiguous pfn range which can then be tried for allocation with *alloc_contig_range()*. This routine is intended for allocation requests which can not be fulfilled with the buddy allocator.

The allocated memory is always aligned to a page boundary. If nr_pages is a power of two, then allocated range is also guaranteed to be aligned to same nr_pages (e.g. 1GB request would be aligned to 1GB).

Allocated pages can be freed with free_contig_range() or by manually calling __free_page() on each allocated page.

Return

pointer to contiguous pages on success, or NULL if not successful.

int **numa_nearest_node**(int node, unsigned int state)
Find nearest node by state

Parameters

int node
: Node id to start the search

unsigned int state
: State to filter the search

Description

Lookup the closest node by distance if **nid** is not in state.

Return

this **node** if it is in state, otherwise the closest node by distance

struct *folio* ***vma_alloc_folio**(gfp_t gfp, int order, struct vm_area_struct *vma, unsigned long addr, bool hugepage)

Allocate a folio for a VMA.

Parameters

gfp_t gfp
: GFP flags.

int order
: Order of the folio.

struct vm_area_struct *vma
: Pointer to VMA or NULL if not available.

unsigned long addr
: Virtual address of the allocation. Must be inside **vma**.

bool hugepage
: For hugepages try only the preferred node if possible.

Description

Allocate a folio for a specific address in **vma**, using the appropriate NUMA policy. When **vma** is not NULL the caller must hold the mmap_lock of the mm_struct of the VMA to prevent it from going away. Should be used for all allocations for folios that will be mapped into user space.

Return

The folio on success or NULL if allocation fails.

struct page ***alloc_pages**(gfp_t gfp, unsigned order)

Allocate pages.

Parameters

gfp_t gfp
: GFP flags.

unsigned order
: Power of two of number of pages to allocate.

Description

Allocate 1 << **order** contiguous pages. The physical address of the first page is naturally aligned (eg an order-3 allocation will be aligned to a multiple of 8 * PAGE_SIZE bytes). The NUMA policy of the current process is honoured when in process context.

Context

Can be called from any context, providing the appropriate GFP flags are used.

Return

The page on success or NULL if allocation fails.

int **mpol_misplaced**(struct *page* *page, struct vm_area_struct *vma, unsigned long addr)
check whether current page node is valid in policy

Parameters

struct page *page
page to be checked

struct vm_area_struct *vma
vm area where page mapped

unsigned long addr
virtual address where page mapped

Description

Lookup current policy node id for vma,addr and "compare to" page's node id. Policy determination "mimics" alloc_page_vma(). Called from fault path where we know the vma and faulting address.

Return

NUMA_NO_NODE if the page is in a node that is valid for this policy, or a suitable node ID to allocate a replacement page from.

void **mpol_shared_policy_init**(struct shared_policy *sp, struct mempolicy *mpol)
initialize shared policy for inode

Parameters

struct shared_policy *sp
pointer to inode shared policy

struct mempolicy *mpol
struct mempolicy to install

Description

Install non-NULL **mpol** in inode's shared policy rb-tree. On entry, the current task has a reference on a non-NULL **mpol**. This must be released on exit. This is called at get_inode() calls and we can use GFP_KERNEL.

int **mpol_parse_str**(char *str, struct mempolicy **mpol)
parse string to mempolicy, for tmpfs mpol mount option.

Parameters

char *str
string containing mempolicy to parse

struct mempolicy **mpol
pointer to struct mempolicy pointer, returned on success.

Description

Format of input:
<mode>[=<flags>][:<nodelist>]

Return

0 on success, else 1

void **mpol_to_str**(char *buffer, int maxlen, struct mempolicy *pol)
format a mempolicy structure for printing

Parameters

char *buffer
to contain formatted mempolicy string

int maxlen
length of **buffer**

struct mempolicy *pol
pointer to mempolicy to be formatted

Description

Convert **pol** into a string. If **buffer** is too short, truncate the string. Recommend a **maxlen** of at least 32 for the longest mode, “interleave”, the longest flag, “relative”, and to display at least a few node ids.

struct **folio**
Represents a contiguous set of bytes.

Definition:

```
struct folio {
    unsigned long flags;
    union {
        struct list_head lru;
        unsigned int mlock_count;
    };
    struct address_space *mapping;
    pgoff_t index;
    union {
        void *private;
        swp_entry_t swap;
    };
    atomic_t _mapcount;
    atomic_t _refcount;
#ifdef CONFIG_MEMCG;
    unsigned long memcg_data;
#endif;
    atomic_t _entire_mapcount;
    atomic_t _nr_pages_mapped;
    atomic_t _pincount;
```

```
#ifdef CONFIG_64BIT;
    unsigned int _folio_nr_pages;
#endif;
    void *_hugetlb_subpool;
    void *_hugetlb_cgroup;
    void *_hugetlb_cgroup_rsvd;
    void *_hugetlb_hwpoison;
    struct list_head _deferred_list;
};
```

Members

flags
Identical to the page flags.

{unnamed_union}
anonymous

lru
Least Recently Used list; tracks how recently this folio was used.

mlock_count
Number of times this folio has been pinned by mlock().

mapping
The file this page belongs to, or refers to the anon_vma for anonymous memory.

index
Offset within the file, in units of pages. For anonymous memory, this is the index from the beginning of the mmap.

{unnamed_union}
anonymous

private
Filesystem per-folio data (see *folio_attach_private()*).

swap
Used for swp_entry_t if folio_test_swapcache().

_mapcount
Do not access this member directly. Use *folio_mapcount()* to find out how many times this folio is mapped by userspace.

_refcount
Do not access this member directly. Use *folio_ref_count()* to find how many references there are to this folio.

memcg_data
Memory Control Group data.

_entire_mapcount
Do not use directly, call folio_entire_mapcount().

_nr_pages_mapped
Do not use directly, call *folio_mapcount()*.

_pincount
Do not use directly, call *folio_maybe_dma_pinned()*.

_folio_nr_pages
Do not use directly, call *folio_nr_pages()*.

_hugetlb_subpool
Do not use directly, use accessor in hugetlb.h.

_hugetlb_cgroup
Do not use directly, use accessor in hugetlb_cgroup.h.

_hugetlb_cgroup_rsvd
Do not use directly, use accessor in hugetlb_cgroup.h.

_hugetlb_hwpoison
Do not use directly, call raw_hwp_list_head().

_deferred_list
Folios to be split under memory pressure.

Description

A folio is a physically, virtually and logically contiguous set of bytes. It is a power-of-two in size, and it is aligned to that same power-of-two. It is at least as large as `PAGE_SIZE`. If it is in the page cache, it is at a file offset which is a multiple of that power-of-two. It may be mapped into userspace at an address which is at an arbitrary page offset, but its kernel virtual address is aligned to its size.

struct **ptdesc**

Memory descriptor for page tables.

Definition:

```
struct ptdesc {
    unsigned long __page_flags;
    union {
        struct rcu_head pt_rcu_head;
        struct list_head pt_list;
        struct {
            unsigned long _pt_pad_1;
            pgtable_t pmd_huge_pte;
        };
    };
    unsigned long __page_mapping;
    union {
        struct mm_struct *pt_mm;
        atomic_t pt_frag_refcount;
    };
    union {
        unsigned long _pt_pad_2;
#if ALLOC_SPLIT_PTLOCKS;
        spinlock_t *ptl;
#else;
        spinlock_t ptl;
#endif;
```

```
    };
    unsigned int __page_type;
    atomic_t _refcount;
#ifdef CONFIG_MEMCG;
    unsigned long pt_memcg_data;
#endif;
};
```

Members

__page_flags
Same as page flags. Unused for page tables.

{unnamed_union}
anonymous

pt_rcu_head
For freeing page table pages.

pt_list
List of used page tables. Used for s390 and x86.

{unnamed_struct}
anonymous

_pt_pad_1
Padding that aliases with page's compound head.

pmd_huge_pte
Protected by ptdesc->ptl, used for THPs.

__page_mapping
Aliases with page->mapping. Unused for page tables.

{unnamed_union}
anonymous

pt_mm
Used for x86 pgds.

pt_frag_refcount
For fragmented page table tracking. Powerpc and s390 only.

{unnamed_union}
anonymous

_pt_pad_2
Padding to ensure proper alignment.

ptl
Lock for the page table.

ptl
Lock for the page table.

__page_type
Same as page->page_type. Unused for page tables.

_refcount
Same as page refcount. Used for s390 page tables.

pt_memcg_data
Memcg data. Tracked for page tables here.

Description

This struct overlays struct page for now. Do not modify without a good understanding of the issues.

type **vm_fault_t**
Return type for page fault handlers.

Description

Page fault handlers return a bitmask of `VM_FAULT` values.

enum **vm_fault_reason**
Page fault handlers return a bitmask of these values to tell the core VM what happened when handling the fault. Used to decide whether a process gets delivered SIGBUS or just gets major/minor fault counters bumped up.

Constants

VM_FAULT_OOM
Out Of Memory

VM_FAULT_SIGBUS
Bad access

VM_FAULT_MAJOR
Page read from storage

VM_FAULT_HWPOISON
Hit poisoned small page

VM_FAULT_HWPOISON_LARGE
Hit poisoned large page. Index encoded in upper bits

VM_FAULT_SIGSEGV
segmentation fault

VM_FAULT_NOPAGE
->fault installed the pte, not return page

VM_FAULT_LOCKED
->fault locked the returned page

VM_FAULT_RETRY
->fault blocked, must retry

VM_FAULT_FALLBACK
huge page fault failed, fall back to small

VM_FAULT_DONE_COW
->fault has fully handled COW

VM_FAULT_NEEDDSYNC
->fault did not modify page tables and needs fsync() to complete (for synchronous page faults in DAX)

VM_FAULT_COMPLETED
: ->fault completed, meanwhile mmap lock released

VM_FAULT_HINDEX_MASK
: mask HINDEX value

enum **fault_flag**

Fault flag definitions.

Constants

FAULT_FLAG_WRITE
: Fault was a write fault.

FAULT_FLAG_MKWRITE
: Fault was mkwrite of existing PTE.

FAULT_FLAG_ALLOW_RETRY
: Allow to retry the fault if blocked.

FAULT_FLAG_RETRY_NOWAIT
: Don't drop mmap_lock and wait when retrying.

FAULT_FLAG_KILLABLE
: The fault task is in SIGKILL killable region.

FAULT_FLAG_TRIED
: The fault has been tried once.

FAULT_FLAG_USER
: The fault originated in userspace.

FAULT_FLAG_REMOTE
: The fault is not for current task/mm.

FAULT_FLAG_INSTRUCTION
: The fault was during an instruction fetch.

FAULT_FLAG_INTERRUPTIBLE
: The fault can be interrupted by non-fatal signals.

FAULT_FLAG_UNSHARE
: The fault is an unsharing request to break COW in a COW mapping, making sure that an exclusive anon page is mapped after the fault.

FAULT_FLAG_ORIG_PTE_VALID
: whether the fault has vmf->orig_pte cached. We should only access orig_pte if this flag set.

FAULT_FLAG_VMA_LOCK
: The fault is handled under VMA lock.

Description

About **FAULT_FLAG_ALLOW_RETRY** and **FAULT_FLAG_TRIED**: we can specify whether we would allow page faults to retry by specifying these two fault flags correctly. Currently there can be three legal combinations:

(a) **ALLOW_RETRY and !TRIED: this means the page fault allows retry, and**
this is the first try

(b) **ALLOW_RETRY and TRIED: this means the page fault allows retry, and** we've already tried at least once

(c) !ALLOW_RETRY and !TRIED: this means the page fault does not allow retry

The unlisted combination (!ALLOW_RETRY && TRIED) is illegal and should never be used. Note that page faults can be allowed to retry for multiple times, in which case we'll have an initial fault with flags (a) then later on continuous faults with flags (b). We should always try to detect pending signals before a retry to make sure the continuous page faults can still be interrupted if necessary.

The combination FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE is illegal. FAULT_FLAG_UNSHARE is ignored and treated like an ordinary read fault when applied to mappings that are not COW mappings.

int **folio_is_file_lru**(struct *folio* *folio)

Should the folio be on a file LRU or anon LRU?

Parameters

struct folio *folio
The folio to test.

Description

We would like to get this info without a page flag, but the state needs to survive until the folio is last deleted from the LRU, which could be as far down as __page_cache_release.

Return

An integer (not a boolean!) used to sort a folio onto the right LRU list and to account folios correctly. 1 if **folio** is a regular filesystem backed page cache folio or a lazily freed anonymous folio (e.g. via MADV_FREE). 0 if **folio** is a normal anonymous folio, a tmpfs folio or otherwise ram or swap backed folio.

void **__folio_clear_lru_flags**(struct *folio* *folio)

Clear page lru flags before releasing a page.

Parameters

struct folio *folio
The folio that was on lru and now has a zero reference.

enum lru_list **folio_lru_list**(struct *folio* *folio)

Which LRU list should a folio be on?

Parameters

struct folio *folio
The folio to test.

Return

The LRU list a folio should be on, as an index into the array of LRU lists.

page_folio

`page_folio (p)`

Converts from page to folio.

Parameters

p
The page.

Description

Every page is part of a folio. This function cannot be called on a NULL pointer.

Context

No reference, nor lock is required on **page**. If the caller does not hold a reference, this call may race with a folio split, so it should re-check the folio still contains this page after gaining a reference on the folio.

Return

The folio which contains this page.

folio_page

`folio_page (folio, n)`

Return a page from a folio.

Parameters

folio
The folio.

n
The page number to return.

Description

n is relative to the start of the folio. This function does not check that the page number lies within **folio**; the caller is presumed to have a reference to the page.

bool **folio_test_uptodate**(struct *folio* *folio)

Is this folio up to date?

Parameters

struct folio *folio
The folio.

Description

The uptodate flag is set on a folio when every byte in the folio is at least as new as the corresponding bytes on storage. Anonymous and CoW folios are always uptodate. If the folio is not uptodate, some of the bytes in it may be; see the is_partially_uptodate() address_space operation.

bool **folio_test_large**(struct *folio* *folio)

Does this folio contain more than one page?

Parameters

struct folio *folio
The folio to test.

Return

True if the folio is larger than one page.

bool **PageHuge**(const struct *page* *page)

Determine if the page belongs to hugetlbfs

Parameters

const struct page *page
The page to test.

Context

Any context.

Return

True for hugetlbfs pages, false for anon pages or pages belonging to other filesystems.

int **page_has_private**(struct *page* *page)

Determine if page has private stuff

Parameters

struct page *page
The page to be checked

Description

Determine if a page has private stuff, indicating that release routines should be invoked upon it.

bool **fault_flag_allow_retry_first**(enum *fault_flag* flags)

check ALLOW_RETRY the first time

Parameters

enum fault_flag flags
Fault flags.

Description

This is mostly used for places where we want to try to avoid taking the mmap_lock for too long a time when waiting for another condition to change, in which case we can try to be polite to release the mmap_lock in the first round to avoid potential starvation of other processes that would also want the mmap_lock.

Return

true if the page fault allows retry and this is the first attempt of the fault handling; false otherwise.

unsigned int **folio_order**(struct *folio* *folio)

The allocation order of a folio.

Parameters

struct folio *folio
The folio.

Description

A folio is composed of 2^order pages. See get_order() for the definition of order.

Return

The order of the folio.

int **page_mapcount**(struct *page* *page)

Number of times this precise page is mapped.

Parameters

struct page *page
The page.

Description

The number of times this page is mapped. If this page is part of a large folio, it includes the number of times this page is mapped as part of that folio.

Will report 0 for pages which cannot be mapped into userspace, eg slab, page tables and similar.

int **folio_mapcount**(struct *folio* *folio)

Calculate the number of mappings of this folio.

Parameters

struct folio *folio
The folio.

Description

A large folio tracks both how many times the entire folio is mapped, and how many times each individual page in the folio is mapped. This function calculates the total number of times the folio is mapped.

Return

The number of times this folio is mapped.

bool **folio_mapped**(struct *folio* *folio)

Is this folio mapped into userspace?

Parameters

struct folio *folio
The folio.

Return

True if any page in this folio is referenced by user page tables.

unsigned int **thp_order**(struct *page* *page)

Order of a transparent huge page.

Parameters

struct page *page
Head page of a transparent huge page.

unsigned long **thp_size**(struct *page* *page)

Size of a transparent huge page.

Parameters

struct page *page
Head page of a transparent huge page.

Return

Number of bytes in this page.

void **folio_get**(struct *folio* *folio)

Increment the reference count on a folio.

Parameters

struct folio *folio
The folio.

Context

May be called in any context, as long as you know that you have a refcount on the folio. If you do not already have one, *folio_try_get()* may be the right interface for you to use.

void **folio_put**(struct *folio* *folio)

Decrement the reference count on a folio.

Parameters

struct folio *folio
The folio.

Description

If the folio's reference count reaches zero, the memory will be released back to the page allocator and may be used by another allocation immediately. Do not access the memory or the *struct folio* after calling *folio_put()* unless you can be sure that it wasn't the last reference.

Context

May be called in process or interrupt context, but not in NMI context. May be called while holding a spinlock.

void **folio_put_refs**(struct *folio* *folio, int refs)

Reduce the reference count on a folio.

Parameters

struct folio *folio
The folio.

int refs
The amount to subtract from the folio's reference count.

Description

If the folio's reference count reaches zero, the memory will be released back to the page allocator and may be used by another allocation immediately. Do not access the memory or the *struct folio* after calling *folio_put_refs()* unless you can be sure that these weren't the last references.

Context

May be called in process or interrupt context, but not in NMI context. May be called while holding a spinlock.

void **folios_put**(struct *folio* **folios, unsigned int nr)

Decrement the reference count on an array of folios.

Parameters

struct folio **folios
The folios.

unsigned int nr
How many folios there are.

Description

Like *folio_put()*, but for an array of folios. This is more efficient than writing the loop yourself as it will optimise the locks which need to be taken if the folios are freed.

Context

May be called in process or interrupt context, but not in NMI context. May be called while holding a spinlock.

unsigned long **folio_pfn**(struct *folio* *folio)

Return the Page Frame Number of a folio.

Parameters

struct folio *folio
The folio.

Description

A folio may contain multiple pages. The pages have consecutive Page Frame Numbers.

Return

The Page Frame Number of the first page in the folio.

bool **folio_maybe_dma_pinned**(struct *folio* *folio)

Report if a folio may be pinned for DMA.

Parameters

struct folio *folio
The folio.

Description

This function checks if a folio has been pinned via a call to a function in the pin_user_pages() family.

For small folios, the return value is partially fuzzy: false is not fuzzy, because it means "definitely not pinned for DMA", but true means "probably pinned for DMA, but possibly a false positive due to having at least GUP_PIN_COUNTING_BIAS worth of normal folio references".

False positives are OK, because: a) it's unlikely for a folio to get that many refcounts, and b) all the callers of this routine are expected to be able to deal gracefully with a false positive.

For large folios, the result will be exactly correct. That's because we have more tracking data available: the _pincount field is used instead of the GUP_PIN_COUNTING_BIAS scheme.

For more information, please see *pin_user_pages() and related calls*.

Return

True, if it is likely that the page has been "dma-pinned". False, if the page is definitely not dma-pinned.

bool **is_zero_page**(const struct *page* *page)

Query if a page is a zero page

Parameters

const struct page *page
The page to query

Description

This returns true if **page** is one of the permanent zero pages.

bool **is_zero_folio**(const struct *folio* *folio)

Query if a folio is a zero page

Parameters

const struct folio *folio
The folio to query

Description

This returns true if **folio** is one of the permanent zero pages.

long **folio_nr_pages**(struct *folio* *folio)

The number of pages in the folio.

Parameters

struct folio *folio
The folio.

Return

A positive power of two.

int **thp_nr_pages**(struct *page* *page)

The number of regular pages in this huge page.

Parameters

struct page *page
The head page of a huge page.

struct *folio* ***folio_next**(struct *folio* *folio)

Move to the next physical folio.

Parameters

struct folio *folio
The folio we're currently operating on.

Description

If you have physically contiguous memory which may span more than one folio (eg a `struct bio_vec`), use this function to move from one folio to the next. Do not use it if the memory is only virtually contiguous as the folios are almost certainly not adjacent to each other. This is the folio equivalent to writing `page++`.

Context

We assume that the folios are refcounted and/or locked at a higher level and do not adjust the reference counts.

Return

The next *`struct folio`*.

unsigned int **`folio_shift`**(struct *folio* *folio)

The size of the memory described by this folio.

Parameters

`struct folio *folio`
The folio.

Description

A folio represents a number of bytes which is a power-of-two in size. This function tells you which power-of-two the folio is. See also folio_size() and *`folio_order()`*.

Context

The caller should have a reference on the folio to prevent it from being split. It is not necessary for the folio to be locked.

Return

The base-2 logarithm of the size of this folio.

size_t **`folio_size`**(struct *folio* *folio)

The number of bytes in a folio.

Parameters

`struct folio *folio`
The folio.

Context

The caller should have a reference on the folio to prevent it from being split. It is not necessary for the folio to be locked.

Return

The number of bytes in this folio.

int **`folio_estimated_sharers`**(struct *folio* *folio)

Estimate the number of sharers of a folio.

Parameters

`struct folio *folio`
The folio.

Description

folio_estimated_sharers() aims to serve as a function to efficiently estimate the number of processes sharing a folio. This is done by looking at the precise mapcount of the first subpage in the folio, and assuming the other subpages are the same. This may not be true for large folios. If you want exact mapcounts for exact calculations, look at *page_mapcount()* or folio_total_mapcount().

Return

The estimated number of processes sharing a folio.

struct *ptdesc* ***pagetable_alloc**(gfp_t gfp, unsigned int order)

Allocate pagetables

Parameters

gfp_t gfp
GFP flags

unsigned int order
desired pagetable order

Description

pagetable_alloc allocates memory for page tables as well as a page table descriptor to describe that memory.

Return

The ptdesc describing the allocated page tables.

void **pagetable_free**(struct *ptdesc* *pt)

Free pagetables

Parameters

struct ptdesc *pt
The page table descriptor

Description

pagetable_free frees the memory of all page tables described by a page table descriptor and the memory for the descriptor itself.

struct vm_area_struct ***vma_lookup**(struct mm_struct *mm, unsigned long addr)

Find a VMA at a specific address

Parameters

struct mm_struct *mm
The process address space.

unsigned long addr
The user address.

Return

The vm_area_struct at the given address, `NULL` otherwise.

bool **vma_is_special_huge**(const struct vm_area_struct *vma)

Are transhuge page-table entries considered special?

Parameters

const struct vm_area_struct *vma

Pointer to the struct vm_area_struct to consider

Description

Whether transhuge page-table entries are considered "special" following the definition in vm_normal_page().

Return

true if transhuge page-table entries should be considered special, false otherwise.

int **seal_check_future_write**(int seals, struct vm_area_struct *vma)

Check for F_SEAL_FUTURE_WRITE flag and handle it

Parameters

int seals

the seals to check

struct vm_area_struct *vma

the vma to operate on

Description

Check whether F_SEAL_FUTURE_WRITE is set; if so, do proper check/handling on the vma flags. Return 0 if check pass, or <0 for errors.

int **folio_ref_count**(const struct *folio* *folio)

The reference count on this folio.

Parameters

const struct folio *folio

The folio.

Description

The refcount is usually incremented by calls to *folio_get()* and decremented by calls to *folio_put()*. Some typical users of the folio refcount:

- Each reference from a page table
- The page cache
- Filesystem private data
- The LRU list
- Pipes
- Direct IO which references this page in the process address space

Return

The number of references to this folio.

bool **folio_try_get**(struct *folio* *folio)

Attempt to increase the refcount on a folio.

Parameters

struct folio *folio
The folio.

Description

If you do not already have a reference to a folio, you can attempt to get one using this function. It may fail if, for example, the folio has been freed since you found a pointer to it, or it is frozen for the purposes of splitting or migration.

Return

True if the reference count was successfully incremented.

bool **folio_try_get_rcu**(struct *folio* *folio)

Attempt to increase the refcount on a folio.

Parameters

struct folio *folio
The folio.

Description

This is a version of *folio_try_get()* optimised for non-SMP kernels. If you are still holding the rcu_read_lock() after looking up the page and know that the page cannot have its refcount decreased to zero in interrupt context, you can use this instead of *folio_try_get()*.

Example users include *get_user_pages_fast()* (as pages are not unmapped from interrupt context) and the page cache lookups (as pages are not truncated from interrupt context). We also know that pages are not frozen in interrupt context for the purposes of splitting or migration.

You can also use this function if you're holding a lock that prevents pages being frozen & removed; eg the i_pages lock for the page cache or the mmap_lock or page table lock for page tables. In this case, it will always succeed, and you could have used a plain *folio_get()*, but it's sometimes more convenient to have a common function called from both locked and RCU-protected contexts.

Return

True if the reference count was successfully incremented.

int **is_highmem**(struct *zone* *zone)

helper function to quickly check if a struct zone is a highmem zone or not. This is an attempt to keep references to ZONE_{DMA/NORMAL/HIGHMEM/etc} in general code to a minimum.

Parameters

struct zone *zone
pointer to struct zone variable

Return

1 for a highmem zone, 0 otherwise

for_each_online_pgdat

`for_each_online_pgdat (pgdat)`

helper macro to iterate over all online nodes

Parameters

pgdat
pointer to a pg_data_t variable

for_each_zone

`for_each_zone (zone)`

helper macro to iterate over all memory zones

Parameters

zone
pointer to struct zone variable

Description

The user only needs to declare the zone variable, for_each_zone fills it in.

struct zoneref ***next_zones_zonelist**(struct zoneref *z, enum zone_type highest_zoneidx, nodemask_t *nodes)

Returns the next zone at or below highest_zoneidx within the allowed nodemask using a cursor within a zonelist as a starting point

Parameters

struct zoneref *z
The cursor used as a starting point for the search

enum zone_type highest_zoneidx
The zone index of the highest zone to return

nodemask_t *nodes
An optional nodemask to filter the zonelist with

Description

This function returns the next zone at or below a given zone index that is within the allowed nodemask using a cursor as the starting point for the search. The zoneref returned is a cursor that represents the current zone being examined. It should be advanced by one before calling next_zones_zonelist again.

Return

the next zone at or below highest_zoneidx within the allowed nodemask using a cursor within a zonelist as a starting point

struct zoneref ***first_zones_zonelist**(struct *zonelist* *zonelist, enum zone_type highest_zoneidx, nodemask_t *nodes)

Returns the first zone at or below highest_zoneidx within the allowed nodemask in a zonelist

Parameters

struct zonelist *zonelist
The zonelist to search for a suitable zone

enum zone_type highest_zoneidx
The zone index of the highest zone to return

nodemask_t *nodes
An optional nodemask to filter the zonelist with

Description

This function returns the first zone at or below a given zone index that is within the allowed nodemask. The zoneref returned is a cursor that can be used to iterate the zonelist with next_zones_zonelist by advancing it by one before calling.

When no eligible zone is found, zoneref->zone is NULL (zoneref itself is never NULL). This may happen either genuinely, or due to concurrent nodemask update due to cpuset modification.

Return

Zoneref pointer for the first suitable zone found

for_each_zone_zonelist_nodemask

`for_each_zone_zonelist_nodemask (zone, z, zlist, highidx, nodemask)`

helper macro to iterate over valid zones in a zonelist at or below a given zone index and within a nodemask

Parameters

zone
The current zone in the iterator

z
The current pointer within zonelist->_zonerefs being iterated

zlist
The zonelist being iterated

highidx
The zone index of the highest zone to return

nodemask
Nodemask allowed by the allocator

Description

This iterator iterates though all zones at or below a given zone index and within a given nodemask

for_each_zone_zonelist

`for_each_zone_zonelist (zone, z, zlist, highidx)`

helper macro to iterate over valid zones in a zonelist at or below a given zone index

Parameters

zone
The current zone in the iterator

z
: The current pointer within zonelist->zones being iterated

zlist
: The zonelist being iterated

highidx
: The zone index of the highest zone to return

Description

This iterator iterates though all zones at or below a given zone index.

int **pfn_valid**(unsigned long pfn)
: check if there is a valid memory map entry for a PFN

Parameters

unsigned long pfn
: the page frame number to check

Description

Check if there is a valid memory map entry aka struct page for the **pfn**. Note, that availability of the memory map entry does not imply that there is actual usable memory at that **pfn**. The struct page may represent a hole or an unusable page frame.

Return

1 for PFNs that have memory map entries and 0 otherwise

struct address_space ***folio_mapping**(struct *folio* *folio)
: Find the mapping where this folio is stored.

Parameters

struct folio *folio
: The folio.

Description

For folios which are in the page cache, return the mapping that this page belongs to. Folios in the swap cache return the swap mapping this page is stored in (which is different from the mapping for the swap file or swap device where the data is stored).

You can call this for folios which aren't in the swap cache or page cache and it will return NULL.

int **__anon_vma_prepare**(struct vm_area_struct *vma)
: attach an anon_vma to a memory region

Parameters

struct vm_area_struct *vma
: the memory region in question

Description

This makes sure the memory mapping described by 'vma' has an 'anon_vma' attached to it, so that we can associate the anonymous pages mapped into it with that anon_vma.

The common case will be that we already have one, which is handled inline by anon_vma_prepare(). But if not we either need to find an adjacent mapping that we can re-use the anon_vma from (very common when the only reason for splitting a vma has been mprotect()), or we allocate a new one.

Anon-vma allocations are very subtle, because we may have optimistically looked up an anon_vma in folio_lock_anon_vma_read() and that may actually touch the rwsem even in the newly allocated vma (it depends on RCU to make sure that the anon_vma isn't actually destroyed).

As a result, we need to do proper anon_vma locking even for the new allocation. At the same time, we do not want to do any locking for the common case of already having an anon_vma.

This must be called with the mmap_lock held for reading.

int **folio_referenced**(struct *folio* *folio, int is_locked, struct mem_cgroup *memcg, unsigned long *vm_flags)

Test if the folio was referenced.

Parameters

struct folio *folio
The folio to test.

int is_locked
Caller holds lock on the folio.

struct mem_cgroup *memcg
target memory cgroup

unsigned long *vm_flags
A combination of all the vma->vm_flags which referenced the folio.

Description

Quick test_and_clear_referenced for all mappings of a folio,

Return

The number of mappings which referenced the folio. Return -1 if the function bailed out due to rmap lock contention.

int **pfn_mkclean_range**(unsigned long pfn, unsigned long nr_pages, pgoff_t pgoff, struct vm_area_struct *vma)

Cleans the PTEs (including PMDs) mapped with range of [**pfn**, **pfn** + **nr_pages**) at the specific offset (**pgoff**) within the **vma** of shared mappings. And since clean PTEs should also be readonly, write protects them too.

Parameters

unsigned long pfn
start pfn.

unsigned long nr_pages
number of physically contiguous pages srarting with **pfn**.

pgoff_t pgoff
page offset that the **pfn** mapped with.

struct vm_area_struct *vma
vma that **pfn** mapped within.

Description

Returns the number of cleaned PTEs (including PMDs).

void **page_move_anon_rmap**(struct *page* *page, struct vm_area_struct *vma)

move a page to our anon_vma

Parameters

struct page *page
the page to move to our anon_vma

struct vm_area_struct *vma
the vma the page belongs to

Description

When a page belongs exclusively to one process after a COW event, that page can be moved into the anon_vma that belongs to just that process, so the rmap code will not search the parent or sibling processes.

void **__page_set_anon_rmap**(struct *folio* *folio, struct *page* *page, struct vm_area_struct *vma, unsigned long address, int exclusive)

set up new anonymous rmap

Parameters

struct folio *folio
Folio which contains page.

struct page *page
Page to add to rmap.

struct vm_area_struct *vma
VM area to add page to.

unsigned long address
User virtual address of the mapping

int exclusive
the page is exclusively owned by the current process

void **__page_check_anon_rmap**(struct *folio* *folio, struct *page* *page, struct vm_area_struct *vma, unsigned long address)

sanity check anonymous rmap addition

Parameters

struct folio *folio
The folio containing **page**.

struct page *page
the page to check the mapping of

struct vm_area_struct *vma
the vm area in which the mapping is added

unsigned long address
the user virtual address mapped

void **page_add_anon_rmap**(struct *page* *page, struct vm_area_struct *vma, unsigned long address, rmap_t flags)

add pte mapping to an anonymous page

Parameters

struct page *page
the page to add the mapping to

struct vm_area_struct *vma
the vm area in which the mapping is added

unsigned long address
the user virtual address mapped

rmap_t flags
the rmap flags

Description

The caller needs to hold the pte lock, and the page must be locked in the anon_vma case: to serialize mapping,index checking after setting, and to ensure that PageAnon is not being upgraded racily to PageKsm (but PageKsm is never downgraded to PageAnon).

void **folio_add_new_anon_rmap**(struct *folio* *folio, struct vm_area_struct *vma, unsigned long address)

Add mapping to a new anonymous folio.

Parameters

struct folio *folio
The folio to add the mapping to.

struct vm_area_struct *vma
the vm area in which the mapping is added

unsigned long address
the user virtual address mapped

Description

Like *page_add_anon_rmap()* but must only be called on *new* folios. This means the inc-and-test can be bypassed. The folio does not have to be locked.

If the folio is large, it is accounted as a THP. As the folio is new, it's assumed to be mapped exclusively by a single process.

void **folio_add_file_rmap_range**(struct *folio* *folio, struct *page* *page, unsigned int nr_pages, struct vm_area_struct *vma, bool compound)

add pte mapping to page range of a folio

Parameters

struct folio *folio
The folio to add the mapping to

struct page *page
The first page to add

unsigned int nr_pages
The number of pages which will be mapped

struct vm_area_struct *vma
the vm area in which the mapping is added

bool compound
charge the page as compound or small page

Description

The page range of folio is defined by [first_page, first_page + nr_pages)

The caller needs to hold the pte lock.

void **page_add_file_rmap**(struct *page* *page, struct vm_area_struct *vma, bool compound)
add pte mapping to a file page

Parameters

struct page *page
the page to add the mapping to

struct vm_area_struct *vma
the vm area in which the mapping is added

bool compound
charge the page as compound or small page

Description

The caller needs to hold the pte lock.

void **page_remove_rmap**(struct *page* *page, struct vm_area_struct *vma, bool compound)
take down pte mapping from a page

Parameters

struct page *page
page to remove mapping from

struct vm_area_struct *vma
the vm area from which the mapping is removed

bool compound
uncharge the page as compound or small page

Description

The caller needs to hold the pte lock.

void **try_to_unmap**(struct *folio* *folio, enum ttu_flags flags)
Try to remove all page table mappings to a folio.

Parameters

struct folio *folio
The folio to unmap.

enum ttu_flags flags
action and flags

Description

Tries to remove all the page table entries which are mapping this folio. It is the caller's responsibility to check if the folio is still mapped if needed (use TTU_SYNC to prevent accounting races).

Context

Caller must hold the folio lock.

void **try_to_migrate**(struct *folio* *folio, enum ttu_flags flags)

try to replace all page table mappings with swap entries

Parameters

struct folio *folio
: the folio to replace page table entries for

enum ttu_flags flags
: action and flags

Description

Tries to remove all the page table entries which are mapping this folio and replace them with special swap entries. Caller must hold the folio lock.

bool **folio_make_device_exclusive**(struct *folio* *folio, struct mm_struct *mm, unsigned long address, void *owner)

Mark the folio exclusively owned by a device.

Parameters

struct folio *folio
: The folio to replace page table entries for.

struct mm_struct *mm
: The mm_struct where the folio is expected to be mapped.

unsigned long address
: Address where the folio is expected to be mapped.

void *owner
: passed to MMU_NOTIFY_EXCLUSIVE range notifier callbacks

Description

Tries to remove all the page table entries which are mapping this folio and replace them with special device exclusive swap entries to grant a device exclusive access to the folio.

Context

Caller must hold the folio lock.

Return

false if the page is still mapped, or if it could not be unmapped from the expected address. Otherwise returns true (success).

int **make_device_exclusive_range**(struct mm_struct *mm, unsigned long start, unsigned long end, struct page **pages, void *owner)

Mark a range for exclusive use by a device

Parameters

struct mm_struct *mm
mm_struct of associated target process

unsigned long start
start of the region to mark for exclusive device access

unsigned long end
end address of region

struct page **pages
returns the pages which were successfully marked for exclusive access

void *owner
passed to MMU_NOTIFY_EXCLUSIVE range notifier to allow filtering

Return

number of pages found in the range by GUP. A page is marked for exclusive access only if the page pointer is non-NULL.

Description

This function finds ptes mapping page(s) to the given address range, locks them and replaces mappings with special swap entries preventing userspace CPU access. On fault these entries are replaced with the original mapping after calling MMU notifiers.

A driver using this to program access from a device must use a mmu notifier critical section to hold a device specific lock during programming. Once programming is complete it should drop the page lock and reference after which point CPU access to the page will revoke the exclusive access.

int **migrate_folio**(struct address_space *mapping, struct *folio* *dst, struct *folio* *src, enum migrate_mode mode)

Simple folio migration.

Parameters

struct address_space *mapping
The address_space containing the folio.

struct folio *dst
The folio to migrate the data to.

struct folio *src
The folio containing the current data.

enum migrate_mode mode
How to migrate the page.

Description

Common logic to directly migrate a single LRU folio suitable for folios that do not use PagePrivate/PagePrivate2.

Folios are locked upon entry and exit.

int **buffer_migrate_folio**(struct address_space *mapping, struct *folio* *dst, struct *folio* *src, enum migrate_mode mode)

Migration function for folios with buffers.

Parameters

struct address_space *mapping
The address space containing **src**.

struct folio *dst
The folio to migrate to.

struct folio *src
The folio to migrate from.

enum migrate_mode mode
How to migrate the folio.

Description

This function can only be used if the underlying filesystem guarantees that no other references to **src** exist. For example attached buffer heads are accessed only under the folio lock. If your filesystem cannot provide this guarantee, *buffer_migrate_folio_norefs()* may be more appropriate.

Return

0 on success or a negative errno on failure.

int **buffer_migrate_folio_norefs**(struct address_space *mapping, struct *folio* *dst, struct *folio* *src, enum migrate_mode mode)

Migration function for folios with buffers.

Parameters

struct address_space *mapping
The address space containing **src**.

struct folio *dst
The folio to migrate to.

struct folio *src
The folio to migrate from.

enum migrate_mode mode
How to migrate the folio.

Description

Like *buffer_migrate_folio()* except that this variant is more careful and checks that there are also no buffer head references. This function is the right one for mappings where buffer heads are directly looked up and referenced (such as block device mappings).

Return

0 on success or a negative errno on failure.

unsigned long **unmapped_area**(struct vm_unmapped_area_info *info)

Find an area between the low_limit and the high_limit with the correct alignment and offset, all from **info**. Note: current->mm is used for the search.

Parameters

struct vm_unmapped_area_info *info
The unmapped area information including the range [low_limit - high_limit), the alignment offset and mask.

Return

A memory address or -ENOMEM.

unsigned long **unmapped_area_topdown**(struct vm_unmapped_area_info *info)
Find an area between the low_limit and the high_limit with the correct alignment and offset at the highest available address, all from **info**. Note: current->mm is used for the search.

Parameters

struct vm_unmapped_area_info *info
The unmapped area information including the range [low_limit - high_limit), the alignment offset and mask.

Return

A memory address or -ENOMEM.

struct vm_area_struct ***find_vma_intersection**(struct mm_struct *mm, unsigned long start_addr, unsigned long end_addr)
Look up the first VMA which intersects the interval

Parameters

struct mm_struct *mm
The process address space.

unsigned long start_addr
The inclusive start user address.

unsigned long end_addr
The exclusive end user address.

Return

The first VMA within the provided range, `NULL` otherwise. Assumes start_addr < end_addr.

struct vm_area_struct ***find_vma**(struct mm_struct *mm, unsigned long addr)
Find the VMA for a given address, or the next VMA.

Parameters

struct mm_struct *mm
The mm_struct to check

unsigned long addr
The address

Return

The VMA associated with addr, or the next VMA. May return `NULL` in the case of no VMA at addr or above.

struct vm_area_struct ***find_vma_prev**(struct mm_struct *mm, unsigned long addr, struct vm_area_struct **pprev)
Find the VMA for a given address, or the next vma and set `pprev` to the previous VMA, if any.

Parameters

struct mm_struct *mm
The mm_struct to check

unsigned long addr
The address

struct vm_area_struct **pprev
The pointer to set to the previous VMA

Description

Note that RCU lock is missing here since the external mmap_lock() is used instead.

Return

The VMA associated with **addr**, or the next vma. May return `NULL` in the case of no vma at addr or above.

void __ref **kmemleak_alloc**(const void *ptr, size_t size, int min_count, gfp_t gfp)
register a newly allocated object

Parameters

const void *ptr
pointer to beginning of the object

size_t size
size of the object

int min_count
minimum number of references to this object. If during memory scanning a number of references less than **min_count** is found, the object is reported as a memory leak. If **min_count** is 0, the object is never reported as a leak. If **min_count** is -1, the object is ignored (not scanned and not reported as a leak)

gfp_t gfp
kmalloc() flags used for kmemleak internal memory allocations

Description

This function is called from the kernel allocators when a new object (memory block) is allocated (kmem_cache_alloc, kmalloc etc.).

void __ref **kmemleak_alloc_percpu**(const void __percpu *ptr, size_t size, gfp_t gfp)
register a newly allocated __percpu object

Parameters

const void __percpu *ptr
__percpu pointer to beginning of the object

size_t size
size of the object

gfp_t gfp
flags used for kmemleak internal memory allocations

Description

This function is called from the kernel percpu allocator when a new object (memory block) is allocated (alloc_percpu).

void __ref **kmemleak_vmalloc**(const struct vm_struct *area, size_t size, gfp_t gfp)
register a newly vmalloc'ed object

Parameters

const struct vm_struct *area
pointer to vm_struct

size_t size
size of the object

gfp_t gfp
__vmalloc() flags used for kmemleak internal memory allocations

Description

This function is called from the vmalloc() kernel allocator when a new object (memory block) is allocated.

void __ref **kmemleak_free**(const void *ptr)
unregister a previously registered object

Parameters

const void *ptr
pointer to beginning of the object

Description

This function is called from the kernel allocators when an object (memory block) is freed (kmem_cache_free, kfree, vfree etc.).

void __ref **kmemleak_free_part**(const void *ptr, size_t size)
partially unregister a previously registered object

Parameters

const void *ptr
pointer to the beginning or inside the object. This also represents the start of the range to be freed

size_t size
size to be unregistered

Description

This function is called when only a part of a memory block is freed (usually from the bootmem allocator).

void __ref **kmemleak_free_percpu**(const void __percpu *ptr)
unregister a previously registered __percpu object

Parameters

const void __percpu *ptr
__percpu pointer to beginning of the object

Description

This function is called from the kernel percpu allocator when an object (memory block) is freed (free_percpu).

void __ref **kmemleak_update_trace**(const void *ptr)

update object allocation stack trace

Parameters

const void *ptr
pointer to beginning of the object

Description

Override the object allocation stack trace for cases where the actual allocation place is not always useful.

void __ref **kmemleak_not_leak**(const void *ptr)

mark an allocated object as false positive

Parameters

const void *ptr
pointer to beginning of the object

Description

Calling this function on an object will cause the memory block to no longer be reported as leak and always be scanned.

void __ref **kmemleak_ignore**(const void *ptr)

ignore an allocated object

Parameters

const void *ptr
pointer to beginning of the object

Description

Calling this function on an object will cause the memory block to be ignored (not scanned and not reported as a leak). This is usually done when it is known that the corresponding block is not a leak and does not contain any references to other allocated memory blocks.

void __ref **kmemleak_scan_area**(const void *ptr, size_t size, gfp_t gfp)

limit the range to be scanned in an allocated object

Parameters

const void *ptr
pointer to beginning or inside the object. This also represents the start of the scan area

size_t size
size of the scan area

gfp_t gfp
kmalloc() flags used for kmemleak internal memory allocations

Description

This function is used when it is known that only certain parts of an object contain references to other objects. Kmemleak will only scan these areas reducing the number false negatives.

void __ref **kmemleak_no_scan**(const void *ptr)

do not scan an allocated object

Parameters

const void *ptr
pointer to beginning of the object

Description

This function notifies kmemleak not to scan the given memory block. Useful in situations where it is known that the given object does not contain any references to other objects. Kmemleak will not scan such objects reducing the number of false negatives.

void __ref **kmemleak_alloc_phys**(phys_addr_t phys, size_t size, gfp_t gfp)

similar to kmemleak_alloc but taking a physical address argument

Parameters

phys_addr_t phys
physical address of the object

size_t size
size of the object

gfp_t gfp
kmalloc() flags used for kmemleak internal memory allocations

void __ref **kmemleak_free_part_phys**(phys_addr_t phys, size_t size)

similar to kmemleak_free_part but taking a physical address argument

Parameters

phys_addr_t phys
physical address if the beginning or inside an object. This also represents the start of the range to be freed

size_t size
size to be unregistered

void __ref **kmemleak_ignore_phys**(phys_addr_t phys)

similar to kmemleak_ignore but taking a physical address argument

Parameters

phys_addr_t phys
physical address of the object

void ***devm_memremap_pages**(struct device *dev, struct dev_pagemap *pgmap)

remap and provide memmap backing for the given resource

Parameters

struct device *dev
hosting device for **res**

struct dev_pagemap *pgmap
pointer to a struct dev_pagemap

Notes

1/ At a minimum the range and type members of pgmap must be initialized
by the caller before passing it to this function

Description

2/ The altmap field may optionally be initialized, in which case
PGMAP_ALTMAP_VALID must be set in pgmap->flags.

3/ The ref field may optionally be provided, in which pgmap->ref must be
'live' on entry and will be killed and reaped at devm_memremap_pages_release() time, or if this routine fails.

4/ range is expected to be a host memory range that could feasibly be
treated as a "System RAM" range, i.e. not a device mmio range, but this is not enforced.

struct dev_pagemap ***get_dev_pagemap**(unsigned long pfn, struct dev_pagemap *pgmap)

take a new live reference on the dev_pagemap for **pfn**

Parameters

unsigned long pfn
page frame number to lookup page_map

struct dev_pagemap *pgmap
optional known pgmap that already has a reference

Description

If **pgmap** is non-NULL and covers **pfn** it will be returned as-is. If **pgmap** is non-NULL but does not cover **pfn** the reference to it will be released.

unsigned long **vma_kernel_pagesize**(struct vm_area_struct *vma)

Page size granularity for this VMA.

Parameters

struct vm_area_struct *vma
The user mapping.

Description

Folios in this VMA will be aligned to, and at least the size of the number of bytes returned by this function.

Return

The default size of the folios allocated when backing a VMA.

void **put_pages_list**(struct list_head *pages)

release a list of pages

Parameters

struct list_head *pages
list of pages threaded on page->lru

Description

Release a list of pages which are strung together on page.lru.

void **folio_add_lru**(struct *folio* *folio)

Add a folio to an LRU list.

Parameters

struct folio *folio
The folio to be added to the LRU.

Description

Queue the folio for addition to the LRU. The decision on whether to add the page to the [in]active [file|anon] list is deferred until the folio_batch is drained. This gives a chance for the caller of *folio_add_lru()* have the folio added to the active list using folio_mark_accessed().

void **folio_add_lru_vma**(struct *folio* *folio, struct vm_area_struct *vma)

Add a folio to the appropate LRU list for this VMA.

Parameters

struct folio *folio
The folio to be added to the LRU.

struct vm_area_struct *vma
VMA in which the folio is mapped.

Description

If the VMA is mlocked, **folio** is added to the unevictable list. Otherwise, it is treated the same way as *folio_add_lru()*.

void **deactivate_file_folio**(struct *folio* *folio)

Deactivate a file folio.

Parameters

struct folio *folio
Folio to deactivate.

Description

This function hints to the VM that **folio** is a good reclaim candidate, for example if its invalidation fails due to the folio being dirty or under writeback.

Context

Caller holds a reference on the folio.

void **folio_mark_lazyfree**(struct *folio* *folio)

make an anon folio lazyfree

Parameters

struct folio *folio
folio to deactivate

Description

folio_mark_lazyfree() moves **folio** to the inactive file list. This is done to accelerate the reclaim of **folio**.

void **release_pages**(release_pages_arg arg, int nr)
batched put_page()

Parameters

release_pages_arg arg
array of pages to release

int nr
number of pages

Description

Decrement the reference count on all the pages in **arg**. If it fell to zero, remove the page from the LRU and free it.

Note that the argument can be an array of pages, encoded pages, or folio pointers. We ignore any encoded bits, and turn any of them into just a folio that gets free'd.

void **folio_batch_remove_exceptionals**(struct folio_batch *fbatch)
Prune non-folios from a batch.

Parameters

struct folio_batch *fbatch
The batch to prune

Description

find_get_entries() fills a batch with both folios and shadow/swap/DAX entries. This function prunes all the non-folio entries from **fbatch** without leaving holes, so that it can be passed on to folio-only batch operations.

void **zpool_register_driver**(struct zpool_driver *driver)
register a zpool implementation.

Parameters

struct zpool_driver *driver
driver to register

int **zpool_unregister_driver**(struct zpool_driver *driver)
unregister a zpool implementation.

Parameters

struct zpool_driver *driver
driver to unregister.

Description

Module usage counting is used to prevent using a driver while/after unloading, so if this is called from module exit function, this should never fail; if called from other than the module exit function, and this returns failure, the driver is in use and must remain available.

bool **zpool_has_pool**(char *type)
Check if the pool driver is available

Parameters

char *type
The type of the zpool to check (e.g. zbud, zsmalloc)

Description

This checks if the **type** pool driver is available. This will try to load the requested module, if needed, but there is no guarantee the module will still be loaded and available immediately after calling. If this returns true, the caller should assume the pool is available, but must be prepared to handle the *zpool_create_pool()* returning failure. However if this returns false, the caller should assume the requested pool type is not available; either the requested pool type module does not exist, or could not be loaded, and calling *zpool_create_pool()* with the pool type will fail.

The **type** string must be null-terminated.

Return

true if **type** pool is available, false if not

struct zpool ***zpool_create_pool**(const char *type, const char *name, gfp_t gfp)
Create a new zpool

Parameters

const char *type
The type of the zpool to create (e.g. zbud, zsmalloc)

const char *name
The name of the zpool (e.g. zram0, zswap)

gfp_t gfp
The GFP flags to use when allocating the pool.

Description

This creates a new zpool of the specified type. The gfp flags will be used when allocating memory, if the implementation supports it. If the ops param is NULL, then the created zpool will not be evictable.

Implementations must guarantee this to be thread-safe.

The **type** and **name** strings must be null-terminated.

Return

New zpool on success, NULL on failure.

void **zpool_destroy_pool**(struct *zpool* *zpool)
Destroy a zpool

Parameters

struct zpool *zpool
The zpool to destroy.

Description

Implementations must guarantee this to be thread-safe, however only when destroying different pools. The same pool should only be destroyed once, and should not be used after it is destroyed.

This destroys an existing zpool. The zpool should not be in use.

const char ***zpool_get_type**(struct *zpool* *zpool)

Get the type of the zpool

Parameters

struct zpool *zpool
The zpool to check

Description

This returns the type of the pool.

Implementations must guarantee this to be thread-safe.

Return

The type of zpool.

bool **zpool_malloc_support_movable**(struct *zpool* *zpool)

Check if the zpool supports allocating movable memory

Parameters

struct zpool *zpool
The zpool to check

Description

This returns if the zpool supports allocating movable memory.

Implementations must guarantee this to be thread-safe.

Return

true if the zpool supports allocating movable memory, false if not

int **zpool_malloc**(struct *zpool* *zpool, size_t size, gfp_t gfp, unsigned long *handle)

Allocate memory

Parameters

struct zpool *zpool
The zpool to allocate from.

size_t size
The amount of memory to allocate.

gfp_t gfp
The GFP flags to use when allocating memory.

unsigned long *handle
Pointer to the handle to set

Description

This allocates the requested amount of memory from the pool. The gfp flags will be used when allocating memory, if the implementation supports it. The provided **handle** will be set to the allocated object handle.

Implementations must guarantee this to be thread-safe.

Return

0 on success, negative value on error.

void **zpool_free**(struct *zpool* *zpool, unsigned long handle)

Free previously allocated memory

Parameters

struct zpool *zpool
The zpool that allocated the memory.

unsigned long handle
The handle to the memory to free.

Description

This frees previously allocated memory. This does not guarantee that the pool will actually free memory, only that the memory in the pool will become available for use by the pool.

Implementations must guarantee this to be thread-safe, however only when freeing different handles. The same handle should only be freed once, and should not be used after freeing.

void ***zpool_map_handle**(struct *zpool* *zpool, unsigned long handle, enum zpool_mapmode mapmode)

Map a previously allocated handle into memory

Parameters

struct zpool *zpool
The zpool that the handle was allocated from

unsigned long handle
The handle to map

enum zpool_mapmode mapmode
How the memory should be mapped

Description

This maps a previously allocated handle into memory. The **mapmode** param indicates to the implementation how the memory will be used, i.e. read-only, write-only, read-write. If the implementation does not support it, the memory will be treated as read-write.

This may hold locks, disable interrupts, and/or preemption, and the *zpool_unmap_handle()* must be called to undo those actions. The code that uses the mapped handle should complete its operations on the mapped handle memory quickly and unmap as soon as possible. As the implementation may use per-cpu data, multiple handles should not be mapped concurrently on any cpu.

Return

A pointer to the handle's mapped memory area.

void **zpool_unmap_handle**(struct *zpool* *zpool, unsigned long handle)

Unmap a previously mapped handle

Parameters

struct zpool *zpool
The zpool that the handle was allocated from

unsigned long handle
The handle to unmap

Description

This unmaps a previously mapped handle. Any locks or other actions that the implementation took in *zpool_map_handle()* will be undone here. The memory area returned from *zpool_map_handle()* should no longer be used after this.

u64 **zpool_get_total_size**(struct *zpool* *zpool)

The total size of the pool

Parameters

struct zpool *zpool
The zpool to check

Description

This returns the total size in bytes of the pool.

Return

Total size of the zpool in bytes.

bool **zpool_can_sleep_mapped**(struct *zpool* *zpool)

Test if zpool can sleep when do mapped.

Parameters

struct zpool *zpool
The zpool to test

Description

Some allocators enter non-preemptible context in ->map() callback (e.g. disable pagefaults) and exit that context in ->unmap(), which limits what we can do with the mapped object. For instance, we cannot wait for asynchronous crypto API to decompress such an object or take mutexes since those will call into the scheduler. This function tells us whether we use such an allocator.

Return

true if zpool can sleep; false otherwise.

struct cgroup_subsys_state ***mem_cgroup_css_from_folio**(struct *folio* *folio)

css of the memcg associated with a folio

Parameters

struct folio *folio
folio of interest

Description

If memcg is bound to the default hierarchy, css of the memcg associated with **folio** is returned. The returned css remains associated with **folio** until it is released.

If memcg is bound to a traditional hierarchy, the css of root_mem_cgroup is returned.

ino_t **page_cgroup_ino**(struct *page* *page)

return inode number of the memcg a page is charged to

Parameters

struct page *page
the page

Description

Look up the closest online ancestor of the memory cgroup **page** is charged to and return its inode number or 0 if **page** is not charged to any cgroup. It is safe to call this function without holding a reference to **page**.

Note, this function is inherently racy, because there is nothing to prevent the cgroup inode from getting torn down and potentially reallocated a moment after *page_cgroup_ino()* returns, so it only should be used by callers that do not care (such as procfs interfaces).

void **__mod_memcg_state**(struct mem_cgroup *memcg, int idx, int val)
update cgroup memory statistics

Parameters

struct mem_cgroup *memcg
the memory cgroup

int idx
the stat item - can be enum memcg_stat_item or enum node_stat_item

int val
delta to add to the counter, can be negative

void **__mod_lruvec_state**(struct *lruvec* *lruvec, enum node_stat_item idx, int val)
update lruvec memory statistics

Parameters

struct lruvec *lruvec
the lruvec

enum node_stat_item idx
the stat item

int val
delta to add to the counter, can be negative

Description

The lruvec is the intersection of the NUMA node and a cgroup. This function updates the all three counters that are affected by a change of state at this level: per-node, per-cgroup, per-lruvec.

void **__count_memcg_events**(struct mem_cgroup *memcg, enum vm_event_item idx, unsigned long count)
account VM events in a cgroup

Parameters

struct mem_cgroup *memcg
the memory cgroup

enum vm_event_item idx
the event item

unsigned long count
the number of events that occurred

struct mem_cgroup ***get_mem_cgroup_from_mm**(struct mm_struct *mm)

Obtain a reference on given mm_struct's memcg.

Parameters

struct mm_struct *mm
mm from which memcg should be extracted. It can be NULL.

Description

Obtain a reference on mm->memcg and returns it if successful. If mm is NULL, then the memcg is chosen as follows: 1) The active memcg, if set. 2) current->mm->memcg, if available 3) root memcg If mem_cgroup is disabled, NULL is returned.

struct mem_cgroup ***mem_cgroup_iter**(struct mem_cgroup *root, struct mem_cgroup *prev, struct mem_cgroup_reclaim_cookie *reclaim)

iterate over memory cgroup hierarchy

Parameters

struct mem_cgroup *root
hierarchy root

struct mem_cgroup *prev
previously returned memcg, NULL on first invocation

struct mem_cgroup_reclaim_cookie *reclaim
cookie for shared reclaim walks, NULL for full walks

Description

Returns references to children of the hierarchy below **root**, or **root** itself, or `NULL` after a full round-trip.

Caller must pass the return value in **prev** on subsequent invocations for reference counting, or use *mem_cgroup_iter_break()* to cancel a hierarchy walk before the round-trip is complete.

Reclaimers can specify a node in **reclaim** to divide up the memcgs in the hierarchy among all concurrent reclaimers operating on the same node.

void **mem_cgroup_iter_break**(struct mem_cgroup *root, struct mem_cgroup *prev)

abort a hierarchy walk prematurely

Parameters

struct mem_cgroup *root
hierarchy root

struct mem_cgroup *prev
last visited hierarchy member as returned by *mem_cgroup_iter()*

void **mem_cgroup_scan_tasks**(struct mem_cgroup *memcg, int (*fn)(struct task_struct*, void*), void *arg)

iterate over tasks of a memory cgroup hierarchy

Parameters

struct mem_cgroup *memcg
hierarchy root

int (*fn)(struct task_struct *, void *)
function to call for each task

void *arg
argument passed to **fn**

Description

This function iterates over tasks attached to **memcg** or to any of its descendants and calls **fn** for each task. If **fn** returns a non-zero value, the function breaks the iteration loop. Otherwise, it will iterate over all tasks and return 0.

This function must not be called for the root memory cgroup.

struct lruvec ***folio_lruvec_lock**(struct *folio* *folio)
Lock the lruvec for a folio.

Parameters

struct folio *folio
Pointer to the folio.

Description

These functions are safe to use under any of the following conditions: - folio locked - folio_test_lru false - *folio_memcg_lock()* - folio frozen (refcount of 0)

Return

The lruvec this folio is on with its lock held.

struct lruvec ***folio_lruvec_lock_irq**(struct *folio* *folio)
Lock the lruvec for a folio.

Parameters

struct folio *folio
Pointer to the folio.

Description

These functions are safe to use under any of the following conditions: - folio locked - folio_test_lru false - *folio_memcg_lock()* - folio frozen (refcount of 0)

Return

The lruvec this folio is on with its lock held and interrupts disabled.

struct lruvec ***folio_lruvec_lock_irqsave**(struct *folio* *folio, unsigned long *flags)
Lock the lruvec for a folio.

Parameters

struct folio *folio
Pointer to the folio.

unsigned long *flags
Pointer to irqsave flags.

Description

These functions are safe to use under any of the following conditions: - folio locked - folio_test_lru false - *folio_memcg_lock()* - folio frozen (refcount of 0)

Return

The lruvec this folio is on with its lock held and interrupts disabled.

void **mem_cgroup_update_lru_size**(struct *lruvec* *lruvec, enum lru_list lru, int zid, int nr_pages)

account for adding or removing an lru page

Parameters

struct lruvec *lruvec
mem_cgroup per zone lru vector

enum lru_list lru
index of lru list the page is sitting on

int zid
zone id of the accounted pages

int nr_pages
positive when adding or negative when removing

Description

This function must be called under lru_lock, just before a page is added to or just after a page is removed from an lru list.

unsigned long **mem_cgroup_margin**(struct mem_cgroup *memcg)

calculate chargeable space of a memory cgroup

Parameters

struct mem_cgroup *memcg
the memory cgroup

Description

Returns the maximum amount of memory **mem** can be charged with, in pages.

void **mem_cgroup_print_oom_context**(struct mem_cgroup *memcg, struct task_struct *p)

Print OOM information relevant to memory controller.

Parameters

struct mem_cgroup *memcg
The memory cgroup that went over limit

struct task_struct *p
Task that is going to be killed

NOTE

memcg and **p**'s mem_cgroup can be different when hierarchy is enabled

void **mem_cgroup_print_oom_meminfo**(struct mem_cgroup *memcg)

Print OOM memory information relevant to memory controller.

Parameters

struct mem_cgroup *memcg
The memory cgroup that went over limit

bool **mem_cgroup_oom_synchronize**(bool handle)

complete memcg OOM handling

Parameters

bool handle
actually kill/wait or just clean up the OOM state

Description

This has to be called at the end of a page fault if the memcg OOM handler was enabled.

Memcg supports userspace OOM handling where failed allocations must sleep on a waitqueue until the userspace task resolves the situation. Sleeping directly in the charge context with all kinds of locks held is not a good idea, instead we remember an OOM state in the task and *mem_cgroup_oom_synchronize()* has to be called at the end of the page fault to complete the OOM handling.

Returns `true` if an ongoing memcg OOM situation was detected and completed, `false` otherwise.

struct mem_cgroup ***mem_cgroup_get_oom_group**(struct task_struct *victim, struct mem_cgroup *oom_domain)

get a memory cgroup to clean up after OOM

Parameters

struct task_struct *victim
task to be killed by the OOM killer

struct mem_cgroup *oom_domain
memcg in case of memcg OOM, NULL in case of system-wide OOM

Description

Returns a pointer to a memory cgroup, which has to be cleaned up by killing all belonging OOM-killable tasks.

Caller has to call mem_cgroup_put() on the returned non-NULL memcg.

void **folio_memcg_lock**(struct *folio* *folio)

Bind a folio to its memcg.

Parameters

struct folio *folio
The folio.

Description

This function prevents unlocked LRU folios from being moved to another cgroup.

It ensures lifetime of the bound memcg. The caller is responsible for the lifetime of the folio.

void **folio_memcg_unlock**(struct *folio* *folio)

Release the binding between a folio and its memcg.

Parameters

struct folio *folio
The folio.

Description

This releases the binding created by *folio_memcg_lock()*. This does not change the accounting of this folio to its memcg, but it does permit others to change it.

bool **consume_stock**(struct mem_cgroup *memcg, unsigned int nr_pages)

Try to consume stocked charge on this cpu.

Parameters

struct mem_cgroup *memcg
memcg to consume from.

unsigned int nr_pages
how many pages to charge.

Description

The charges will only happen if **memcg** matches the current cpu's memcg stock, and at least **nr_pages** are available in that stock. Failure to service an allocation will refill the stock.

returns true if successful, false otherwise.

int **__memcg_kmem_charge_page**(struct *page* *page, gfp_t gfp, int order)

charge a kmem page to the current memory cgroup

Parameters

struct page *page
page to charge

gfp_t gfp
reclaim mode

int order
allocation order

Description

Returns 0 on success, an error code on failure.

void **__memcg_kmem_uncharge_page**(struct *page* *page, int order)

uncharge a kmem page

Parameters

struct page *page
page to uncharge

int order
allocation order

int **mem_cgroup_move_swap_account**(swp_entry_t entry, struct mem_cgroup *from, struct mem_cgroup *to)

move swap charge and swap_cgroup's record.

Parameters

swp_entry_t entry
swap entry to be moved

struct mem_cgroup *from
mem_cgroup which the entry is moved from

struct mem_cgroup *to
mem_cgroup which the entry is moved to

Description

It succeeds only when the swap_cgroup's record for this entry is the same as the mem_cgroup's id of **from**.

Returns 0 on success, -EINVAL on failure.

The caller must have charged to **to**, IOW, called page_counter_charge() about both res and memsw, and called css_get().

void **mem_cgroup_wb_stats**(struct bdi_writeback *wb, unsigned long *pfilepages, unsigned long *pheadroom, unsigned long *pdirty, unsigned long *pwriteback)

retrieve writeback related stats from its memcg

Parameters

struct bdi_writeback *wb
bdi_writeback in question

unsigned long *pfilepages
out parameter for number of file pages

unsigned long *pheadroom
out parameter for number of allocatable pages according to memcg

unsigned long *pdirty
out parameter for number of dirty pages

unsigned long *pwriteback
out parameter for number of pages under writeback

Description

Determine the numbers of file, headroom, dirty, and writeback pages in **wb**'s memcg. File, dirty and writeback are self-explanatory. Headroom is a bit more involved.

A memcg's headroom is "min(max, high) - used". In the hierarchy, the headroom is calculated as the lowest headroom of itself and the ancestors. Note that this doesn't consider the actual amount of available memory in the system. The caller should further cap ***pheadroom** accordingly.

struct mem_cgroup ***mem_cgroup_from_id**(unsigned short id)

look up a memcg from a memcg id

Parameters

unsigned short id
the memcg id to look up

Description

Caller must hold rcu_read_lock().

void **mem_cgroup_css_reset**(struct cgroup_subsys_state *css)

reset the states of a mem_cgroup

Parameters

struct cgroup_subsys_state *css
the target css

Description

Reset the states of the mem_cgroup associated with **css**. This is invoked when the userland requests disabling on the default hierarchy but the memcg is pinned through dependency. The memcg should stop applying policies and should revert to the vanilla state as it may be made visible again.

The current implementation only resets the essential configurations. This needs to be expanded to cover all the visible parts.

int **mem_cgroup_move_account**(struct *page* *page, bool compound, struct mem_cgroup *from, struct mem_cgroup *to)

move account of the page

Parameters

struct page *page
the page

bool compound
charge the page as compound or small page

struct mem_cgroup *from
mem_cgroup which the page is moved from.

struct mem_cgroup *to
mem_cgroup which the page is moved to. **from** != **to**.

Description

The page must be locked and not on the LRU.

This function doesn't do "charge" to new cgroup and doesn't do "uncharge" from old cgroup.

enum mc_target_type **get_mctgt_type**(struct vm_area_struct *vma, unsigned long addr, pte_t ptent, union mc_target *target)

get target type of moving charge

Parameters

struct vm_area_struct *vma
the vma the pte to be checked belongs

unsigned long addr
the address corresponding to the pte to be checked

pte_t ptent
the pte to be checked

union mc_target *target
the pointer the target page or swap ent will be stored(can be NULL)

Context

Called with pte lock held.

Return

- MC_TARGET_NONE - If the pte is not a target for move charge.
- MC_TARGET_PAGE - If the page corresponding to this pte is a target for move charge. If **target** is not NULL, the page is stored in target->page with extra refcnt taken (Caller should release it).
- MC_TARGET_SWAP - If the swap entry corresponding to this pte is a target for charge migration. If **target** is not NULL, the entry is stored in target->ent.
- MC_TARGET_DEVICE - Like MC_TARGET_PAGE but page is device memory and thus not on the lru. For now such page is charged like a regular page would be as it is just special memory taking the place of a regular page. See Documentations/vm/hmm.txt and include/linux/hmm.h

void **mem_cgroup_calculate_protection**(struct mem_cgroup *root, struct mem_cgroup *memcg)

check if memory consumption is in the normal range

Parameters

struct mem_cgroup *root
the top ancestor of the sub-tree being checked

struct mem_cgroup *memcg
the memory cgroup to check

Description

WARNING: This function is not stateless! It can only be used as part
of a top-down tree iteration, not for isolated queries.

int **mem_cgroup_swapin_charge_folio**(struct *folio* *folio, struct mm_struct *mm, gfp_t gfp, swp_entry_t entry)

Charge a newly allocated folio for swapin.

Parameters

struct folio *folio
folio to charge.

struct mm_struct *mm
mm context of the victim

gfp_t gfp
reclaim mode

swp_entry_t entry
swap entry for which the folio is allocated

Description

This function charges a folio allocated for swapin. Please call this before adding the folio to the swapcache.

Returns 0 on success. Otherwise, an error code is returned.

void **__mem_cgroup_uncharge_list**(struct list_head *page_list)
uncharge a list of page

Parameters

struct list_head *page_list
list of pages to uncharge

Description

Uncharge a list of pages previously charged with __mem_cgroup_charge().

void **mem_cgroup_migrate**(struct *folio* *old, struct *folio* *new)
Charge a folio's replacement.

Parameters

struct folio *old
Currently circulating folio.

struct folio *new
Replacement folio.

Description

Charge **new** as a replacement folio for **old**. **old** will be uncharged upon free.

Both folios must be locked, **new->mapping** must be set up.

bool **mem_cgroup_charge_skmem**(struct mem_cgroup *memcg, unsigned int nr_pages, gfp_t gfp_mask)
charge socket memory

Parameters

struct mem_cgroup *memcg
memcg to charge

unsigned int nr_pages
number of pages to charge

gfp_t gfp_mask
reclaim mode

Description

Charges **nr_pages** to **memcg**. Returns `true` if the charge fit within **memcg**'s configured limit, `false` if it doesn't.

void **mem_cgroup_uncharge_skmem**(struct mem_cgroup *memcg, unsigned int nr_pages)
uncharge socket memory

Parameters

struct mem_cgroup *memcg
memcg to uncharge

unsigned int nr_pages
number of pages to uncharge

void **mem_cgroup_swapout**(struct *folio* *folio, swp_entry_t entry)
transfer a memsw charge to swap

Parameters

struct folio *folio
folio whose memsw charge to transfer

swp_entry_t entry
swap entry to move the charge to

Description

Transfer the memsw charge of **folio** to **entry**.

int **__mem_cgroup_try_charge_swap**(struct *folio* *folio, swp_entry_t entry)
try charging swap space for a folio

Parameters

struct folio *folio
folio being added to swap

swp_entry_t entry
swap entry to charge

Description

Try to charge **folio**'s memcg for the swap space at **entry**.

Returns 0 on success, -ENOMEM on failure.

void **__mem_cgroup_uncharge_swap**(swp_entry_t entry, unsigned int nr_pages)
uncharge swap space

Parameters

swp_entry_t entry
swap entry to uncharge

unsigned int nr_pages
the amount of swap space to uncharge

bool **obj_cgroup_may_zswap**(struct obj_cgroup *objcg)
check if this cgroup can zswap

Parameters

struct obj_cgroup *objcg
the object cgroup

Description

Check if the hierarchical zswap limit has been reached.

This doesn't check for specific headroom, and it is not atomic either. But with zswap, the size of the allocation is only known once compression has occured, and this optimistic pre-check avoids spending cycles on compression when there is already no room left or zswap is disabled altogether somewhere in the hierarchy.

void **obj_cgroup_charge_zswap**(struct obj_cgroup *objcg, size_t size)
charge compression backend memory

Parameters

struct obj_cgroup *objcg
the object cgroup

size_t size
size of compressed object

Description

This forces the charge after *obj_cgroup_may_zswap()* allowed compression and storage in zwap for this cgroup to go ahead.

void **obj_cgroup_uncharge_zswap**(struct obj_cgroup *objcg, size_t size)
uncharge compression backend memory

Parameters

struct obj_cgroup *objcg
the object cgroup

size_t size
size of compressed object

Description

Uncharges zswap memory on page in.

void **shmem_recalc_inode**(struct *inode* *inode, long alloced, long swapped)
recalculate the block usage of an inode

Parameters

struct inode *inode
inode to recalc

long alloced
the change in number of pages allocated to inode

long swapped
the change in number of pages swapped from inode

Description

We have to calculate the free blocks since the mm can drop undirtied hole pages behind our back.

But normally info->alloced == inode->i_mapping->nrpages + info->swapped So mm freed is info->alloced - (inode->i_mapping->nrpages + info->swapped)

struct file ***shmem_kernel_file_setup**(const char *name, loff_t size, unsigned long flags)
get an unlinked file living in tmpfs which must be kernel internal. There will be NO LSM permission checks against the underlying inode. So users of this interface must do LSM checks at a higher layer. The users are the big_key and shm implementations. LSM checks are provided at the key or shm level rather than the inode.

Parameters

const char *name
name for dentry (to be seen in /proc/<pid>/maps

loff_t size
size to be set for the file

unsigned long flags
VM_NORESERVE suppresses pre-accounting of the entire object size

struct file ***shmem_file_setup**(const char *name, loff_t size, unsigned long flags)
get an unlinked file living in tmpfs

Parameters

const char *name
name for dentry (to be seen in /proc/<pid>/maps

loff_t size
size to be set for the file

unsigned long flags
VM_NORESERVE suppresses pre-accounting of the entire object size

struct file ***shmem_file_setup_with_mnt**(struct vfsmount *mnt, const char *name, loff_t size, unsigned long flags)
get an unlinked file living in tmpfs

Parameters

struct vfsmount *mnt
the tmpfs mount where the file will be created

const char *name
name for dentry (to be seen in /proc/<pid>/maps

loff_t size
size to be set for the file

unsigned long flags
VM_NORESERVE suppresses pre-accounting of the entire object size

int **shmem_zero_setup**(struct vm_area_struct *vma)
setup a shared anonymous mapping

Parameters

struct vm_area_struct *vma
the vma to be mmapped is prepared by do_mmap

struct *folio* ***shmem_read_folio_gfp**(struct address_space *mapping, pgoff_t index, gfp_t gfp)
read into page cache, using specified page allocation flags.

Parameters

struct address_space *mapping
the folio's address_space

pgoff_t index
the folio index

gfp_t gfp
the page allocator flags to use if allocating

Description

This behaves as a tmpfs "read_cache_page_gfp(mapping, index, gfp)", with any new page allocations done using the specified allocation flags. But *read_cache_page_gfp()* uses the ->read_folio() method: which does not suit tmpfs, since it may have pages in swapcache, and needs to find those for itself; although drivers/gpu/drm i915 and ttm rely upon this support.

i915_gem_object_get_pages_gtt() mixes __GFP_NORETRY | __GFP_NOWARN in with the mapping_gfp_mask(), to avoid OOMing the machine unnecessarily.

int **migrate_vma_setup**(struct migrate_vma *args)
prepare to migrate a range of memory

Parameters

struct migrate_vma *args
contains the vma, start, and pfns arrays for the migration

Return

negative errno on failures, 0 when 0 or more pages were migrated without an error.

Description

Prepare to migrate a range of memory virtual address range by collecting all the pages backing each virtual address in the range, saving them inside the src array. Then lock those pages and unmap them. Once the pages are locked and unmapped, check whether each page is pinned or not. Pages that aren't pinned have the MIGRATE_PFN_MIGRATE flag set (by this function) in the corresponding src array entry. Then restores any pages that are pinned, by remapping and unlocking those pages.

The caller should then allocate destination memory and copy source memory to it for all those entries (ie with MIGRATE_PFN_VALID and MIGRATE_PFN_MIGRATE flag set). Once these are allocated and copied, the caller must update each corresponding entry in the dst array with the pfn value of the destination page and with MIGRATE_PFN_VALID. Destination pages must be locked via lock_page().

Note that the caller does not have to migrate all the pages that are marked with MIGRATE_PFN_MIGRATE flag in src array unless this is a migration from device memory to system memory. If the caller cannot migrate a device page back to system memory, then it must return VM_FAULT_SIGBUS, which has severe consequences for the userspace process, so it must be avoided if at all possible.

For empty entries inside CPU page table (pte_none() or pmd_none() is true) we do set MIGRATE_PFN_MIGRATE flag inside the corresponding source array thus allowing the caller to allocate device memory for those unbacked virtual addresses. For this the caller simply has to allocate device memory and properly set the destination entry like for regular migration. Note that this can still fail, and thus inside the device driver you must check if the migration was successful for those entries after calling *migrate_vma_pages()*, just like for regular migration.

After that, the callers must call *migrate_vma_pages()* to go over each entry in the src array that has the MIGRATE_PFN_VALID and MIGRATE_PFN_MIGRATE flag set. If the corresponding entry in dst array has MIGRATE_PFN_VALID flag set, then *migrate_vma_pages()* to migrate struct page information from the source struct page to the destination struct page. If it fails to migrate the struct page information, then it clears the MIGRATE_PFN_MIGRATE flag in the src array.

At this point all successfully migrated pages have an entry in the src array with MIGRATE_PFN_VALID and MIGRATE_PFN_MIGRATE flag set and the dst array entry with MIGRATE_PFN_VALID flag set.

Once *migrate_vma_pages()* returns the caller may inspect which pages were successfully migrated, and which were not. Successfully migrated pages will have the MIGRATE_PFN_MIGRATE flag set for their src array entry.

It is safe to update device page table after *migrate_vma_pages()* because both destination and source page are still locked, and the mmap_lock is held in read mode (hence no one can unmap the range being migrated).

Once the caller is done cleaning up things and updating its page table (if it chose to do so, this is not an obligation) it finally calls *migrate_vma_finalize()* to update the CPU page table to point to new pages for successfully migrated pages or otherwise restore the CPU page table to point to the original source pages.

void **migrate_device_pages**(unsigned long *src_pfns, unsigned long *dst_pfns, unsigned long npages)

migrate meta-data from src page to dst page

Parameters

unsigned long *src_pfns
src_pfns returned from *migrate_device_range()*

unsigned long *dst_pfns
array of pfns allocated by the driver to migrate memory to

unsigned long npages
number of pages in the range

Description

Equivalent to *migrate_vma_pages()*. This is called to migrate struct page meta-data from source struct page to destination.

void **migrate_vma_pages**(struct migrate_vma *migrate)

migrate meta-data from src page to dst page

Parameters

struct migrate_vma *migrate
migrate struct containing all migration information

Description

This migrates struct page meta-data from source struct page to destination struct page. This effectively finishes the migration from source page to the destination page.

void **migrate_vma_finalize**(struct migrate_vma *migrate)

restore CPU page table entry

Parameters

struct migrate_vma *migrate
migrate struct containing all migration information

Description

This replaces the special migration pte entry with either a mapping to the new page if migration was successful for that page, or to the original page otherwise.

This also unlocks the pages and puts them back on the lru, or drops the extra refcount, for device pages.

int **migrate_device_range**(unsigned long *src_pfns, unsigned long start, unsigned long npages)

migrate device private pfns to normal memory.

Parameters

unsigned long *src_pfns
array large enough to hold migrating source device private pfns.

unsigned long start
starting pfn in the range to migrate.

unsigned long npages
number of pages to migrate.

Description

migrate_vma_setup() is similar in concept to *migrate_vma_setup()* except that instead of looking up pages based on virtual address mappings a range of device pfns that should be migrated to system memory is used instead.

This is useful when a driver needs to free device memory but doesn't know the virtual mappings of every page that may be in device memory. For example this is often the case when a driver is being unloaded or unbound from a device.

Like *migrate_vma_setup()* this function will take a reference and lock any migrating pages that aren't free before unmapping them. Drivers may then allocate destination pages and start copying data from the device to CPU memory before calling *migrate_device_pages()*.

struct **wp_walk**

Private struct for pagetable walk callbacks

Definition:

```
struct wp_walk {
    struct mmu_notifier_range range;
    unsigned long tlbflush_start;
    unsigned long tlbflush_end;
    unsigned long total;
};
```

Members

range
Range for mmu notifiers

tlbflush_start
Address of first modified pte

tlbflush_end
Address of last modified pte + 1

total
    Total number of modified ptes

int **wp_pte**(pte_t *pte, unsigned long addr, unsigned long end, struct mm_walk *walk)
    Write-protect a pte

Parameters

pte_t *pte
    Pointer to the pte

unsigned long addr
    The start of protecting virtual address

unsigned long end
    The end of protecting virtual address

struct mm_walk *walk
    pagetable walk callback argument

Description

The function write-protects a pte and records the range in virtual address space of touched ptes for efficient range TLB flushes.

struct **clean_walk**
    Private struct for the clean_record_pte function.

Definition:

```
struct clean_walk {
    struct wp_walk base;
    pgoff_t bitmap_pgoff;
    unsigned long *bitmap;
    pgoff_t start;
    pgoff_t end;
};
```

Members

base
    struct wp_walk we derive from

bitmap_pgoff
    Address_space Page offset of the first bit in **bitmap**

bitmap
    Bitmap with one bit for each page offset in the address_space range covered.

start
    Address_space page offset of first modified pte relative to **bitmap_pgoff**

end
    Address_space page offset of last modified pte relative to **bitmap_pgoff**

int **clean_record_pte**(pte_t *pte, unsigned long addr, unsigned long end, struct mm_walk *walk)
    Clean a pte and record its address space offset in a bitmap

Parameters

pte_t *pte
Pointer to the pte

unsigned long addr
The start of virtual address to be clean

unsigned long end
The end of virtual address to be clean

struct mm_walk *walk
pagetable walk callback argument

Description

The function cleans a pte and records the range in virtual address space of touched ptes for efficient TLB flushes. It also records dirty ptes in a bitmap representing page offsets in the address_space, as well as the first and last of the bits touched.

unsigned long **wp_shared_mapping_range**(struct address_space *mapping, pgoff_t first_index, pgoff_t nr)

Write-protect all ptes in an address space range

Parameters

struct address_space *mapping
The address_space we want to write protect

pgoff_t first_index
The first page offset in the range

pgoff_t nr
Number of incremental page offsets to cover

Note

This function currently skips transhuge page-table entries, since it's intended for dirty-tracking on the PTE level. It will warn on encountering transhuge write-enabled entries, though, and can easily be extended to handle them as well.

Return

The number of ptes actually write-protected. Note that already write-protected ptes are not counted.

unsigned long **clean_record_shared_mapping_range**(struct address_space *mapping, pgoff_t first_index, pgoff_t nr, pgoff_t bitmap_pgoff, unsigned long *bitmap, pgoff_t *start, pgoff_t *end)

Clean and record all ptes in an address space range

Parameters

struct address_space *mapping
The address_space we want to clean

pgoff_t first_index
The first page offset in the range

pgoff_t nr
Number of incremental page offsets to cover

pgoff_t bitmap_pgoff
The page offset of the first bit in **bitmap**

unsigned long *bitmap
Pointer to a bitmap of at least **nr** bits. The bitmap needs to cover the whole range **first_index**..**first_index** + **nr**.

pgoff_t *start
Pointer to number of the first set bit in **bitmap**. is modified as new bits are set by the function.

pgoff_t *end
Pointer to the number of the last set bit in **bitmap**. none set. The value is modified as new bits are set by the function.

Description

When this function returns there is no guarantee that a CPU has not already dirtied new ptes. However it will not clean any ptes not reported in the bitmap. The guarantees are as follows:

- All ptes dirty when the function starts executing will end up recorded in the bitmap.
- All ptes dirtied after that will either remain dirty, be recorded in the bitmap or both.

If a caller needs to make sure all dirty ptes are picked up and none additional are added, it first needs to write-protect the address-space range and make sure new writers are blocked in page_mkwrite() or pfn_mkwrite(). And then after a TLB flush following the write-protection pick up all dirty bits.

This function currently skips transhuge page-table entries, since it's intended for dirty-tracking on the PTE level. It will warn on encountering transhuge dirty entries, though, and can easily be extended to handle them as well.

Return

The number of dirty ptes actually cleaned.

bool **pcpu_addr_in_chunk**(struct pcpu_chunk *chunk, void *addr)
check if the address is served from this chunk

Parameters

struct pcpu_chunk *chunk
chunk of interest

void *addr
percpu address

Return

True if the address is served from this chunk.

bool **pcpu_check_block_hint**(struct pcpu_block_md *block, int bits, size_t align)
check against the contig hint

Parameters

struct pcpu_block_md *block
block of interest

int bits
size of allocation

size_t align
alignment of area (max PAGE_SIZE)

Description

Check to see if the allocation can fit in the block's contig hint. Note, a chunk uses the same hints as a block so this can also check against the chunk's contig hint.

void **pcpu_next_md_free_region**(struct pcpu_chunk *chunk, int *bit_off, int *bits)
finds the next hint free area

Parameters

struct pcpu_chunk *chunk
chunk of interest

int *bit_off
chunk offset

int *bits
size of free area

Description

Helper function for pcpu_for_each_md_free_region. It checks block->contig_hint and performs aggregation across blocks to find the next hint. It modifies bit_off and bits in-place to be consumed in the loop.

void **pcpu_next_fit_region**(struct pcpu_chunk *chunk, int alloc_bits, int align, int *bit_off, int *bits)
finds fit areas for a given allocation request

Parameters

struct pcpu_chunk *chunk
chunk of interest

int alloc_bits
size of allocation

int align
alignment of area (max PAGE_SIZE)

int *bit_off
chunk offset

int *bits
size of free area

Description

Finds the next free region that is viable for use with a given size and alignment. This only returns if there is a valid area to be used for this allocation. block->first_free is returned if the allocation request fits within the block to see if the request can be fulfilled prior to the contig hint.

void *`pcpu_mem_zalloc`(size_t size, gfp_t gfp)
allocate memory

Parameters

size_t size
bytes to allocate

gfp_t gfp
allocation flags

Description

Allocate **size** bytes. If **size** is smaller than PAGE_SIZE, kzalloc() is used; otherwise, the equivalent of vzalloc() is used. This is to facilitate passing through whitelisted flags. The returned memory is always zeroed.

Return

Pointer to the allocated area on success, NULL on failure.

void `pcpu_mem_free`(void *ptr)
free memory

Parameters

void *ptr
memory to free

Description

Free **ptr**. **ptr** should have been allocated using *pcpu_mem_zalloc()*.

void `pcpu_chunk_relocate`(struct pcpu_chunk *chunk, int oslot)
put chunk in the appropriate chunk slot

Parameters

struct pcpu_chunk *chunk
chunk of interest

int oslot
the previous slot it was on

Description

This function is called after an allocation or free changed **chunk**. New slot according to the changed state is determined and **chunk** is moved to the slot. Note that the reserved chunk is never put on chunk slots.

Context

pcpu_lock.

void `pcpu_block_update`(struct pcpu_block_md *block, int start, int end)
updates a block given a free area

Parameters

struct pcpu_block_md *block
block of interest

int start
start offset in block

int end
end offset in block

Description

Updates a block given a known free area. The region [start, end) is expected to be the entirety of the free area within a block. Chooses the best starting offset if the contig hints are equal.

void **pcpu_chunk_refresh_hint**(struct pcpu_chunk *chunk, bool full_scan)
updates metadata about a chunk

Parameters

struct pcpu_chunk *chunk
chunk of interest

bool full_scan
if we should scan from the beginning

Description

Iterates over the metadata blocks to find the largest contig area. A full scan can be avoided on the allocation path as this is triggered if we broke the contig_hint. In doing so, the scan_hint will be before the contig_hint or after if the scan_hint == contig_hint. This cannot be prevented on freeing as we want to find the largest area possibly spanning blocks.

void **pcpu_block_refresh_hint**(struct pcpu_chunk *chunk, int index)

Parameters

struct pcpu_chunk *chunk
chunk of interest

int index
index of the metadata block

Description

Scans over the block beginning at first_free and updates the block metadata accordingly.

void **pcpu_block_update_hint_alloc**(struct pcpu_chunk *chunk, int bit_off, int bits)
update hint on allocation path

Parameters

struct pcpu_chunk *chunk
chunk of interest

int bit_off
chunk offset

int bits
size of request

Description

Updates metadata for the allocation path. The metadata only has to be refreshed by a full scan iff the chunk's contig hint is broken. Block level scans are required if the block's contig hint is broken.

void **pcpu_block_update_hint_free**(struct pcpu_chunk *chunk, int bit_off, int bits)
updates the block hints on the free path

Parameters

struct pcpu_chunk *chunk
chunk of interest

int bit_off
chunk offset

int bits
size of request

Description

Updates metadata for the allocation path. This avoids a blind block refresh by making use of the block contig hints. If this fails, it scans forward and backward to determine the extent of the free area. This is capped at the boundary of blocks.

A chunk update is triggered if a page becomes free, a block becomes free, or the free spans across blocks. This tradeoff is to minimize iterating over the block metadata to update chunk_md->contig_hint. chunk_md->contig_hint may be off by up to a page, but it will never be more than the available space. If the contig hint is contained in one block, it will be accurate.

bool **pcpu_is_populated**(struct pcpu_chunk *chunk, int bit_off, int bits, int *next_off)
determines if the region is populated

Parameters

struct pcpu_chunk *chunk
chunk of interest

int bit_off
chunk offset

int bits
size of area

int *next_off
return value for the next offset to start searching

Description

For atomic allocations, check if the backing pages are populated.

Return

Bool if the backing pages are populated. next_index is to skip over unpopulated blocks in pcpu_find_block_fit.

int **pcpu_find_block_fit**(struct pcpu_chunk *chunk, int alloc_bits, size_t align, bool pop_only)
finds the block index to start searching

Parameters

struct pcpu_chunk *chunk
chunk of interest

int alloc_bits
size of request in allocation units

size_t align
alignment of area (max PAGE_SIZE bytes)

bool pop_only
use populated regions only

Description

Given a chunk and an allocation spec, find the offset to begin searching for a free region. This iterates over the bitmap metadata blocks to find an offset that will be guaranteed to fit the requirements. It is not quite first fit as if the allocation does not fit in the contig hint of a block or chunk, it is skipped. This errs on the side of caution to prevent excess iteration. Poor alignment can cause the allocator to skip over blocks and chunks that have valid free areas.

Return

The offset in the bitmap to begin searching. -1 if no offset is found.

int **pcpu_alloc_area**(struct pcpu_chunk *chunk, int alloc_bits, size_t align, int start)
allocates an area from a pcpu_chunk

Parameters

struct pcpu_chunk *chunk
chunk of interest

int alloc_bits
size of request in allocation units

size_t align
alignment of area (max PAGE_SIZE)

int start
bit_off to start searching

Description

This function takes in a **start** offset to begin searching to fit an allocation of **alloc_bits** with alignment **align**. It needs to scan the allocation map because if it fits within the block's contig hint, **start** will be block->first_free. This is an attempt to fill the allocation prior to breaking the contig hint. The allocation and boundary maps are updated accordingly if it confirms a valid free area.

Return

Allocated addr offset in **chunk** on success. -1 if no matching area is found.

int **pcpu_free_area**(struct pcpu_chunk *chunk, int off)
frees the corresponding offset

Parameters

struct pcpu_chunk *chunk
chunk of interest

int off
addr offset into chunk

Description

This function determines the size of an allocation to free using the boundary bitmap and clears the allocation map.

Return

Number of freed bytes.

struct pcpu_chunk ***pcpu_alloc_first_chunk**(unsigned long tmp_addr, int map_size)
creates chunks that serve the first chunk

Parameters

unsigned long tmp_addr
the start of the region served

int map_size
size of the region served

Description

This is responsible for creating the chunks that serve the first chunk. The base_addr is page aligned down of **tmp_addr** while the region end is page aligned up. Offsets are kept track of to determine the region served. All this is done to appease the bitmap allocator in avoiding partial blocks.

Return

Chunk serving the region at **tmp_addr** of **map_size**.

void **pcpu_chunk_populated**(struct pcpu_chunk *chunk, int page_start, int page_end)
post-population bookkeeping

Parameters

struct pcpu_chunk *chunk
pcpu_chunk which got populated

int page_start
the start page

int page_end
the end page

Description

Pages in [**page_start**,**page_end**) have been populated to **chunk**. Update the bookkeeping information accordingly. Must be called after each successful population.

void **pcpu_chunk_depopulated**(struct pcpu_chunk *chunk, int page_start, int page_end)
post-depopulation bookkeeping

Parameters

struct pcpu_chunk *chunk
pcpu_chunk which got depopulated

int page_start
the start page

int page_end
the end page

Description

Pages in [**page_start**,**page_end**) have been depopulated from **chunk**. Update the bookkeeping information accordingly. Must be called after each successful depopulation.

struct pcpu_chunk ***pcpu_chunk_addr_search**(void *addr)
determine chunk containing specified address

Parameters

void *addr
address for which the chunk needs to be determined.

Description

This is an internal function that handles all but static allocations. Static percpu address values should never be passed into the allocator.

Return

The address of the found chunk.

void __percpu ***pcpu_alloc**(size_t size, size_t align, bool reserved, gfp_t gfp)
the percpu allocator

Parameters

size_t size
size of area to allocate in bytes

size_t align
alignment of area (max PAGE_SIZE)

bool reserved
allocate from the reserved chunk if available

gfp_t gfp
allocation flags

Description

Allocate percpu area of **size** bytes aligned at **align**. If **gfp** doesn't contain `GFP_KERNEL`, the allocation is atomic. If **gfp** has __GFP_NOWARN then no warning will be triggered on invalid or failed allocation requests.

Return

Percpu pointer to the allocated area on success, NULL on failure.

void __percpu ***__alloc_percpu_gfp**(size_t size, size_t align, gfp_t gfp)
allocate dynamic percpu area

Parameters

size_t size
size of area to allocate in bytes

size_t align
alignment of area (max PAGE_SIZE)

gfp_t gfp
allocation flags

Description

Allocate zero-filled percpu area of **size** bytes aligned at **align**. If **gfp** doesn't contain GFP_KERNEL, the allocation doesn't block and can be called from any context but is a lot more likely to fail. If **gfp** has __GFP_NOWARN then no warning will be triggered on invalid or failed allocation requests.

Return

Percpu pointer to the allocated area on success, NULL on failure.

void __percpu ***__alloc_percpu**(size_t size, size_t align)
allocate dynamic percpu area

Parameters

size_t size
size of area to allocate in bytes

size_t align
alignment of area (max PAGE_SIZE)

Description

Equivalent to __alloc_percpu_gfp(size, align, GFP_KERNEL).

void __percpu ***__alloc_reserved_percpu**(size_t size, size_t align)
allocate reserved percpu area

Parameters

size_t size
size of area to allocate in bytes

size_t align
alignment of area (max PAGE_SIZE)

Description

Allocate zero-filled percpu area of **size** bytes aligned at **align** from reserved percpu area if arch has set it up; otherwise, allocation is served from the same dynamic area. Might sleep. Might trigger writeouts.

Context

Does GFP_KERNEL allocation.

Return

Percpu pointer to the allocated area on success, NULL on failure.

void **pcpu_balance_free**(bool empty_only)
manage the amount of free chunks

Parameters

bool empty_only
free chunks only if there are no populated pages

Description

If empty_only is `false`, reclaim all fully free chunks regardless of the number of populated pages. Otherwise, only reclaim chunks that have no populated pages.

Context

pcpu_lock (can be dropped temporarily)

void **pcpu_balance_populated**(void)

manage the amount of populated pages

Parameters

void
no arguments

Description

Maintain a certain amount of populated pages to satisfy atomic allocations. It is possible that this is called when physical memory is scarce causing OOM killer to be triggered. We should avoid doing so until an actual allocation causes the failure as it is possible that requests can be serviced from already backed regions.

Context

pcpu_lock (can be dropped temporarily)

void **pcpu_reclaim_populated**(void)

scan over to_depopulate chunks and free empty pages

Parameters

void
no arguments

Description

Scan over chunks in the depopulate list and try to release unused populated pages back to the system. Depopulated chunks are sidelined to prevent repopulating these pages unless required. Fully free chunks are reintegrated and freed accordingly (1 is kept around). If we drop below the empty populated pages threshold, reintegrate the chunk if it has empty free pages. Each chunk is scanned in the reverse order to keep populated pages close to the beginning of the chunk.

Context

pcpu_lock (can be dropped temporarily)

void **pcpu_balance_workfn**(struct work_struct *work)

manage the amount of free chunks and populated pages

Parameters

struct work_struct *work
unused

Description

For each chunk type, manage the number of fully free chunks and the number of populated pages. An important thing to consider is when pages are freed and how they contribute to the global counts.

void **free_percpu**(void __percpu *ptr)
free percpu area

Parameters

void __percpu *ptr
pointer to area to free

Description

Free percpu area **ptr**.

Context

Can be called from atomic context.

bool **is_kernel_percpu_address**(unsigned long addr)
test whether address is from static percpu area

Parameters

unsigned long addr
address to test

Description

Test whether **addr** belongs to in-kernel static percpu area. Module static percpu areas are not considered. For those, use is_module_percpu_address().

Return

`true` if **addr** is from in-kernel static percpu area, `false` otherwise.

phys_addr_t **per_cpu_ptr_to_phys**(void *addr)
convert translated percpu address to physical address

Parameters

void *addr
the address to be converted to physical address

Description

Given **addr** which is dereferenceable address obtained via one of percpu access macros, this function translates it into its physical address. The caller is responsible for ensuring **addr** stays valid until this function finishes.

percpu allocator has special setup for the first chunk, which currently supports either embedding in linear address space or vmalloc mapping, and, from the second one, the backing allocator (currently either vm or km) provides translation.

The addr can be translated simply without checking if it falls into the first chunk. But the current code reflects better how percpu allocator actually works, and the verification can discover both bugs in percpu allocator itself and *per_cpu_ptr_to_phys()* callers. So we keep current code.

Return

The physical address for **addr**.

struct pcpu_alloc_info ***pcpu_alloc_alloc_info**(int nr_groups, int nr_units)
allocate percpu allocation info

Parameters

int nr_groups
the number of groups

int nr_units
the number of units

Description

Allocate ai which is large enough for **nr_groups** groups containing **nr_units** units. The returned ai's groups[0].cpu_map points to the cpu_map array which is long enough for **nr_units** and filled with NR_CPUS. It's the caller's responsibility to initialize cpu_map pointer of other groups.

Return

Pointer to the allocated pcpu_alloc_info on success, NULL on failure.

void **pcpu_free_alloc_info**(struct pcpu_alloc_info *ai)
free percpu allocation info

Parameters

struct pcpu_alloc_info *ai
pcpu_alloc_info to free

Description

Free **ai** which was allocated by *pcpu_alloc_alloc_info()*.

void **pcpu_dump_alloc_info**(const char *lvl, const struct pcpu_alloc_info *ai)
print out information about pcpu_alloc_info

Parameters

const char *lvl
loglevel

const struct pcpu_alloc_info *ai
allocation info to dump

Description

Print out information about **ai** using loglevel **lvl**.

void **pcpu_setup_first_chunk**(const struct pcpu_alloc_info *ai, void *base_addr)
initialize the first percpu chunk

Parameters

const struct pcpu_alloc_info *ai
pcpu_alloc_info describing how to percpu area is shaped

void *base_addr
mapped address

Description

Initialize the first percpu chunk which contains the kernel static percpu area. This function is to be called from arch percpu area setup path.

ai contains all information necessary to initialize the first chunk and prime the dynamic percpu allocator.

ai->static_size is the size of static percpu area.

ai->reserved_size, if non-zero, specifies the amount of bytes to reserve after the static area in the first chunk. This reserves the first chunk such that it's available only through reserved percpu allocation. This is primarily used to serve module percpu static areas on architectures where the addressing model has limited offset range for symbol relocations to guarantee module percpu symbols fall inside the relocatable range.

ai->dyn_size determines the number of bytes available for dynamic allocation in the first chunk. The area between **ai->static_size** + **ai->reserved_size** + **ai->dyn_size** and **ai->unit_size** is unused.

ai->unit_size specifies unit size and must be aligned to PAGE_SIZE and equal to or larger than **ai->static_size** + **ai->reserved_size** + **ai->dyn_size**.

ai->atom_size is the allocation atom size and used as alignment for vm areas.

ai->alloc_size is the allocation size and always multiple of **ai->atom_size**. This is larger than **ai->atom_size** if **ai->unit_size** is larger than **ai->atom_size**.

ai->nr_groups and **ai->groups** describe virtual memory layout of percpu areas. Units which should be colocated are put into the same group. Dynamic VM areas will be allocated according to these groupings. If **ai->nr_groups** is zero, a single group containing all units is assumed.

The caller should have mapped the first chunk at **base_addr** and copied static data to each unit.

The first chunk will always contain a static and a dynamic region. However, the static region is not managed by any chunk. If the first chunk also contains a reserved region, it is served by two chunks - one for the reserved region and one for the dynamic region. They share the same vm, but use offset regions in the area allocation map. The chunk serving the dynamic region is circulated in the chunk slots and available for dynamic allocation like any other chunk.

struct pcpu_alloc_info ***pcpu_build_alloc_info**(size_t reserved_size, size_t dyn_size, size_t atom_size, pcpu_fc_cpu_distance_fn_t cpu_distance_fn)

build alloc_info considering distances between CPUs

Parameters

size_t reserved_size
the size of reserved percpu area in bytes

size_t dyn_size
minimum free size for dynamic allocation in bytes

size_t atom_size
allocation atom size

pcpu_fc_cpu_distance_fn_t cpu_distance_fn
callback to determine distance between cpus, optional

Description

This function determines grouping of units, their mappings to cpus and other parameters considering needed percpu size, allocation atom size and distances between CPUs.

Groups are always multiples of atom size and CPUs which are of LOCAL_DISTANCE both ways are grouped together and share space for units in the same group. The returned configuration is guaranteed to have CPUs on different nodes on different groups and >=75% usage of allocated virtual address space.

Return

On success, pointer to the new allocation_info is returned. On failure, ERR_PTR value is returned.

int **pcpu_embed_first_chunk**(size_t reserved_size, size_t dyn_size, size_t atom_size, pcpu_fc_cpu_distance_fn_t cpu_distance_fn, pcpu_fc_cpu_to_node_fn_t cpu_to_nd_fn)

embed the first percpu chunk into bootmem

Parameters

size_t reserved_size
the size of reserved percpu area in bytes

size_t dyn_size
minimum free size for dynamic allocation in bytes

size_t atom_size
allocation atom size

pcpu_fc_cpu_distance_fn_t cpu_distance_fn
callback to determine distance between cpus, optional

pcpu_fc_cpu_to_node_fn_t cpu_to_nd_fn
callback to convert cpu to it's node, optional

Description

This is a helper to ease setting up embedded first percpu chunk and can be called where *pcpu_setup_first_chunk()* is expected.

If this function is used to setup the first chunk, it is allocated by calling pcpu_fc_alloc and used as-is without being mapped into vmalloc area. Allocations are always whole multiples of **atom_size** aligned to **atom_size**.

This enables the first chunk to piggy back on the linear physical mapping which often uses larger page size. Please note that this can result in very sparse cpu->unit mapping on NUMA machines thus requiring large vmalloc address space. Don't use this allocator if vmalloc space is not orders of magnitude larger than distances between node memory addresses (ie. 32bit NUMA machines).

dyn_size specifies the minimum dynamic area size.

If the needed size is smaller than the minimum or specified unit size, the leftover is returned using pcpu_fc_free.

Return

0 on success, -errno on failure.

int **pcpu_page_first_chunk**(size_t reserved_size, pcpu_fc_cpu_to_node_fn_t cpu_to_nd_fn)

map the first chunk using PAGE_SIZE pages

Parameters

size_t reserved_size
the size of reserved percpu area in bytes

pcpu_fc_cpu_to_node_fn_t cpu_to_nd_fn
callback to convert cpu to it's node, optional

Description

This is a helper to ease setting up page-remapped first percpu chunk and can be called where *pcpu_setup_first_chunk()* is expected.

This is the basic allocator. Static percpu area is allocated page-by-page into vmalloc area.

Return

0 on success, -errno on failure.

long **copy_from_user_nofault**(void *dst, const void __user *src, size_t size)
safely attempt to read from a user-space location

Parameters

void *dst
pointer to the buffer that shall take the data

const void __user *src
address to read from. This must be a user address.

size_t size
size of the data chunk

Description

Safely read from user address **src** to the buffer at **dst**. If a kernel fault happens, handle that and return -EFAULT.

long **copy_to_user_nofault**(void __user *dst, const void *src, size_t size)
safely attempt to write to a user-space location

Parameters

void __user *dst
address to write to

const void *src
pointer to the data that shall be written

size_t size
size of the data chunk

Description

Safely write to address **dst** from the buffer at **src**. If a kernel fault happens, handle that and return -EFAULT.

long **strncpy_from_user_nofault**(char *dst, const void __user *unsafe_addr, long count)

- Copy a NUL terminated string from unsafe user address.

Parameters

char *dst
Destination address, in kernel space. This buffer must be at least **count** bytes long.

const void __user *unsafe_addr
Unsafe user address.

long count
Maximum number of bytes to copy, including the trailing NUL.

Description

Copies a NUL-terminated string from unsafe user address to kernel buffer.

On success, returns the length of the string INCLUDING the trailing NUL.

If access fails, returns -EFAULT (some data may have been copied and the trailing NUL added).

If **count** is smaller than the length of the string, copies **count**-1 bytes, sets the last byte of **dst** buffer to NUL and returns **count**.

long **strnlen_user_nofault**(const void __user *unsafe_addr, long count)

- Get the size of a user string INCLUDING final NUL.

Parameters

const void __user *unsafe_addr
The string to measure.

long count
Maximum count (including NUL)

Description

Get the size of a NUL-terminated string in user space without pagefault.

Returns the size of the string INCLUDING the terminating NUL.

If the string is too long, returns a number larger than **count**. User has to check the return value against "> count". On exception (or invalid count), returns 0.

Unlike strnlen_user, this can be used from IRQ handler etc. because it disables pagefaults.

bool **writeback_throttling_sane**(struct scan_control *sc)
is the usual dirty throttling mechanism available?

Parameters

struct scan_control *sc
scan_control in question

Description

The normal page dirty throttling mechanism in balance_dirty_pages() is completely broken with the legacy memcg and direct stalling in shrink_folio_list() is used for throttling instead, which lacks all the niceties such as fairness, adaptive pausing, bandwidth proportional allocation and configurability.

This function tests whether the vmscan currently in progress can assume that the normal dirty throttling mechanism is operational.

unsigned long **lruvec_lru_size**(struct *lruvec* *lruvec, enum lru_list lru, int zone_idx)
Returns the number of pages on the given LRU list.

Parameters

struct lruvec *lruvec
lru vector

enum lru_list lru
lru to use

int zone_idx
zones to consider (use MAX_NR_ZONES - 1 for the whole LRU list)

void **synchronize_shrinkers**(void)
Wait for all running shrinkers to complete.

Parameters

void
no arguments

Description

This is equivalent to calling unregister_shrink() and register_shrinker(), but atomically and with less overhead. This is useful to guarantee that all shrinker invocations have seen an update, before freeing memory, similar to rcu.

unsigned long **shrink_slab**(gfp_t gfp_mask, int nid, struct mem_cgroup *memcg, int priority)
shrink slab caches

Parameters

gfp_t gfp_mask
allocation context

int nid
node whose slab caches to target

struct mem_cgroup *memcg
memory cgroup whose slab caches to target

int priority
the reclaim priority

Description

Call the shrink functions to age shrinkable caches.

nid is passed along to shrinkers with SHRINKER_NUMA_AWARE set, unaware shrinkers will receive a node id of 0 instead.

memcg specifies the memory cgroup to target. Unaware shrinkers are called only if it is the root cgroup.

priority is sc->priority, we take the number of objects and >> by priority in order to get the scan target.

Returns the number of reclaimed slab objects.

long **remove_mapping**(struct address_space *mapping, struct *folio* *folio)
Attempt to remove a folio from its mapping.

Parameters

struct address_space *mapping
The address space.

struct folio *folio
The folio to remove.

Description

If the folio is dirty, under writeback or if someone else has a ref on it, removal will fail.

Return

The number of pages removed from the mapping. 0 if the folio could not be removed.

Context

The caller should have a single refcount on the folio and hold its lock.

void **folio_putback_lru**(struct *folio* *folio)
Put previously isolated folio onto appropriate LRU list.

Parameters

struct folio *folio
Folio to be returned to an LRU list.

Description

Add previously isolated **folio** to appropriate LRU list. The folio may still be unevictable for other reasons.

Context

lru_lock must not be held, interrupts must be enabled.

bool **folio_isolate_lru**(struct *folio* *folio)
Try to isolate a folio from its LRU list.

Parameters

struct folio *folio
Folio to isolate from its LRU list.

Description

Isolate a **folio** from an LRU list and adjust the vmstat statistic corresponding to whatever LRU list the folio was on.

The folio will have its LRU flag cleared. If it was found on the active list, it will have the Active flag set. If it was found on the unevictable list, it will have the Unevictable flag set. These flags may need to be cleared by the caller before letting the page go.

(1) Must be called with an elevated refcount on the folio. This is a fundamental difference from isolate_lru_folios() (which is called without a stable reference).

(2) The lru_lock must not be held.

(3) Interrupts must be enabled.

Context

Return

true if the folio was removed from an LRU list. false if the folio was not on an LRU list.

void **check_move_unevictable_folios**(struct folio_batch *fbatch)
Move evictable folios to appropriate zone lru list

Parameters

struct folio_batch *fbatch
Batch of lru folios to check.

Description

Checks folios for evictability, if an evictable folio is in the unevictable lru list, moves it to the appropriate evictable lru list. This function should be only used for lru folios.

void **__remove_pages**(unsigned long pfn, unsigned long nr_pages, struct vmem_altmap *altmap)
remove sections of pages

Parameters

unsigned long pfn
starting pageframe (must be aligned to start of a section)

unsigned long nr_pages
number of pages to remove (must be multiple of section size)

struct vmem_altmap *altmap
alternative device page map or `NULL` if default memmap is used

Description

Generic helper function to remove section mappings and sysfs entries for the section of the memory we are removing. Caller needs to make sure that pages are marked reserved and zones are adjust properly by calling offline_pages().

void **try_offline_node**(int nid)

Parameters

int nid
the node ID

Description

Offline a node if all memory sections and cpus of the node are removed.

NOTE

The caller must call lock_device_hotplug() to serialize hotplug and online/offline operations before this call.

void **__remove_memory**(u64 start, u64 size)
Remove memory if every memory block is offline

Parameters

u64 start
physical address of the region to remove

u64 size
size of the region to remove

NOTE

The caller must call lock_device_hotplug() to serialize hotplug and online/offline operations before this call, as required by *try_offline_node()*.

unsigned long **mmu_interval_read_begin**(struct mmu_interval_notifier *interval_sub)

Begin a read side critical section against a VA range

Parameters

struct mmu_interval_notifier *interval_sub
The interval subscription

Description

mmu_iterval_read_begin()/mmu_iterval_read_retry() implement a collision-retry scheme similar to seqcount for the VA range under subscription. If the mm invokes invalidation during the critical section then mmu_interval_read_retry() will return true.

This is useful to obtain shadow PTEs where teardown or setup of the SPTEs require a blocking context. The critical region formed by this can sleep, and the required 'user_lock' can also be a sleeping lock.

The caller is required to provide a 'user_lock' to serialize both teardown and setup.

The return value should be passed to mmu_interval_read_retry().

int **mmu_notifier_register**(struct mmu_notifier *subscription, struct mm_struct *mm)

Register a notifier on a mm

Parameters

struct mmu_notifier *subscription
The notifier to attach

struct mm_struct *mm
The mm to attach the notifier to

Description

Must not hold mmap_lock nor any other VM related lock when calling this registration function. Must also ensure mm_users can't go down to zero while this runs to avoid races with mmu_notifier_release, so mm has to be current->mm or the mm should be pinned safely such as with get_task_mm(). If the mm is not current->mm, the mm_users pin should be released by calling mmput after mmu_notifier_register returns.

mmu_notifier_unregister() or *mmu_notifier_put()* must be always called to unregister the notifier.

While the caller has a mmu_notifier get the subscription->mm pointer will remain valid, and can be converted to an active mm pointer via mmget_not_zero().

struct mmu_notifier ***mmu_notifier_get_locked**(const struct mmu_notifier_ops *ops, struct mm_struct *mm)

Return the single struct mmu_notifier for the mm & ops

Parameters

const struct mmu_notifier_ops *ops
The operations struct being subscribe with

struct mm_struct *mm
The mm to attach notifiers too

Description

This function either allocates a new mmu_notifier via ops->alloc_notifier(), or returns an already existing notifier on the list. The value of the ops pointer is used to determine when two notifiers are the same.

Each call to mmu_notifier_get() must be paired with a call to *mmu_notifier_put()*. The caller must hold the write side of mm->mmap_lock.

While the caller has a mmu_notifier get the mm pointer will remain valid, and can be converted to an active mm pointer via mmget_not_zero().

void **mmu_notifier_put**(struct mmu_notifier *subscription)
Release the reference on the notifier

Parameters

struct mmu_notifier *subscription
The notifier to act on

Description

This function must be paired with each mmu_notifier_get(), it releases the reference obtained by the get. If this is the last reference then process to free the notifier will be run asynchronously.

Unlike mmu_notifier_unregister() the get/put flow only calls ops->release when the mm_struct is destroyed. Instead free_notifier is always called to release any resources held by the user.

As ops->release is not guaranteed to be called, the user must ensure that all sptes are dropped, and no new sptes can be established before *mmu_notifier_put()* is called.

This function can be called from the ops->release callback, however the caller must still ensure it is called pairwise with mmu_notifier_get().

Modules calling this function must call *mmu_notifier_synchronize()* in their __exit functions to ensure the async work is completed.

int **mmu_interval_notifier_insert**(struct mmu_interval_notifier *interval_sub, struct mm_struct *mm, unsigned long start, unsigned long length, const struct mmu_interval_notifier_ops *ops)
Insert an interval notifier

Parameters

struct mmu_interval_notifier *interval_sub
Interval subscription to register

struct mm_struct *mm
mm_struct to attach to

unsigned long start
Starting virtual address to monitor

unsigned long length
Length of the range to monitor

const struct mmu_interval_notifier_ops *ops
Interval notifier operations to be called on matching events

Description

This function subscribes the interval notifier for notifications from the mm. Upon return the ops related to mmu_interval_notifier will be called whenever an event that intersects with the given range occurs.

Upon return the range_notifier may not be present in the interval tree yet. The caller must use the normal interval notifier read flow via *mmu_interval_read_begin()* to establish SPTEs for this range.

void **mmu_interval_notifier_remove**(struct mmu_interval_notifier *interval_sub)

Remove a interval notifier

Parameters

struct mmu_interval_notifier *interval_sub
Interval subscription to unregister

Description

This function must be paired with *mmu_interval_notifier_insert()*. It cannot be called from any ops callback.

Once this returns ops callbacks are no longer running on other CPUs and will not be called in future.

void **mmu_notifier_synchronize**(void)

Ensure all mmu_notifiers are freed

Parameters

void
no arguments

Description

This function ensures that all outstanding async SRU work from *mmu_notifier_put()* is completed. After it returns any mmu_notifier_ops associated with an unused mmu_notifier will no longer be called.

Before using the caller must ensure that all of its mmu_notifiers have been fully released via *mmu_notifier_put()*.

Modules using the *mmu_notifier_put()* API should call this in their __exit function to avoid module unloading races.

size_t **balloon_page_list_enqueue**(struct balloon_dev_info *b_dev_info, struct list_head *pages)

inserts a list of pages into the balloon page list.

Parameters

struct balloon_dev_info *b_dev_info
balloon device descriptor where we will insert a new page to

struct list_head *pages
pages to enqueue - allocated using balloon_page_alloc.

Description

Driver must call this function to properly enqueue balloon pages before definitively removing them from the guest system.

Return

number of pages that were enqueued.

size_t **balloon_page_list_dequeue**(struct balloon_dev_info *b_dev_info, struct list_head *pages, size_t n_req_pages)

removes pages from balloon's page list and returns a list of the pages.

Parameters

struct balloon_dev_info *b_dev_info
balloon device descriptor where we will grab a page from.

struct list_head *pages
pointer to the list of pages that would be returned to the caller.

size_t n_req_pages
number of requested pages.

Description

Driver must call this function to properly de-allocate a previous enlisted balloon pages before definitively releasing it back to the guest system. This function tries to remove **n_req_pages** from the ballooned pages and return them to the caller in the **pages** list.

Note that this function may fail to dequeue some pages even if the balloon isn't empty - since the page list can be temporarily empty due to compaction of isolated pages.

Return

number of pages that were added to the **pages** list.

vm_fault_t **vmf_insert_pfn_pmd**(struct vm_fault *vmf, pfn_t pfn, bool write)

insert a pmd size pfn

Parameters

struct vm_fault *vmf
Structure describing the fault

pfn_t pfn
pfn to insert

bool write
whether it's a write fault

Description

Insert a pmd size pfn. See *vmf_insert_pfn()* for additional info.

Return

vm_fault_t value.

vm_fault_t **vmf_insert_pfn_pud**(struct vm_fault *vmf, pfn_t pfn, bool write)

insert a pud size pfn

Parameters

struct vm_fault *vmf
Structure describing the fault

pfn_t pfn
pfn to insert

bool write
whether it's a write fault

Description

Insert a pud size pfn. See *vmf_insert_pfn()* for additional info.

Return

vm_fault_t value.

int **io_mapping_map_user**(struct io_mapping *iomap, struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, unsigned long size)

remap an I/O mapping to userspace

Parameters

struct io_mapping *iomap
the source io_mapping

struct vm_area_struct *vma
user vma to map to

unsigned long addr
target user address to start at

unsigned long pfn
physical address of kernel memory

unsigned long size
size of map area

Note

this is only safe if the mm semaphore is held when called.

6.8 The genalloc/genpool subsystem

There are a number of memory-allocation subsystems in the kernel, each aimed at a specific need. Sometimes, however, a kernel developer needs to implement a new allocator for a specific range of special-purpose memory; often that memory is located on a device somewhere. The author of the driver for that device can certainly write a little allocator to get the job done, but that is the way to fill the kernel with dozens of poorly tested allocators. Back in 2005, Jes Sorensen lifted one of those allocators from the sym53c8xx_2 driver and posted it as a generic module for the creation of ad hoc memory allocators. This code was merged for the 2.6.13 release; it has been modified considerably since then.

Code using this allocator should include <linux/genalloc.h>. The action begins with the creation of a pool using one of:

struct gen_pool *`gen_pool_create`(int min_alloc_order, int nid)

create a new special memory pool

Parameters

int min_alloc_order
log base 2 of number of bytes each bitmap bit represents

int nid
node id of the node the pool structure should be allocated on, or -1

Description

Create a new special memory pool that can be used to manage special purpose memory not managed by the regular kmalloc/kfree interface.

struct gen_pool *`devm_gen_pool_create`(struct device *dev, int min_alloc_order, int nid, const char *name)

managed gen_pool_create

Parameters

struct device *dev
device that provides the gen_pool

int min_alloc_order
log base 2 of number of bytes each bitmap bit represents

int nid
node selector for allocated gen_pool, `NUMA_NO_NODE` for all nodes

const char *name
name of a gen_pool or NULL, identifies a particular gen_pool on device

Description

Create a new special memory pool that can be used to manage special purpose memory not managed by the regular kmalloc/kfree interface. The pool will be automatically destroyed by the device management code.

A call to gen_pool_create() will create a pool. The granularity of allocations is set with min_alloc_order; it is a log-base-2 number like those used by the page allocator, but it refers to bytes rather than pages. So, if min_alloc_order is passed as 3, then all allocations will be a multiple of eight bytes. Increasing min_alloc_order decreases the memory required to track the memory in the pool. The nid parameter specifies which NUMA node should be used for the allocation of the housekeeping structures; it can be -1 if the caller doesn't care.

The "managed" interface devm_gen_pool_create() ties the pool to a specific device. Among other things, it will automatically clean up the pool when the given device is destroyed.

A pool is shut down with:

void `gen_pool_destroy`(struct gen_pool *pool)

destroy a special memory pool

Parameters

struct gen_pool *pool
pool to destroy

Description

Destroy the specified special memory pool. Verifies that there are no outstanding allocations.

It's worth noting that, if there are still allocations outstanding from the given pool, this function will take the rather extreme step of invoking BUG(), crashing the entire system. You have been warned.

A freshly created pool has no memory to allocate. It is fairly useless in that state, so one of the first orders of business is usually to add memory to the pool. That can be done with one of:

int **gen_pool_add**(struct gen_pool *pool, unsigned long addr, size_t size, int nid)
add a new chunk of special memory to the pool

Parameters

struct gen_pool *pool
pool to add new memory chunk to

unsigned long addr
starting address of memory chunk to add to pool

size_t size
size in bytes of the memory chunk to add to pool

int nid
node id of the node the chunk structure and bitmap should be allocated on, or -1

Description

Add a new chunk of special memory to the specified pool.

Returns 0 on success or a -ve errno on failure.

int **gen_pool_add_owner**(struct gen_pool *pool, unsigned long virt, phys_addr_t phys, size_t size, int nid, void *owner)
add a new chunk of special memory to the pool

Parameters

struct gen_pool *pool
pool to add new memory chunk to

unsigned long virt
virtual starting address of memory chunk to add to pool

phys_addr_t phys
physical starting address of memory chunk to add to pool

size_t size
size in bytes of the memory chunk to add to pool

int nid
node id of the node the chunk structure and bitmap should be allocated on, or -1

void *owner
private data the publisher would like to recall at alloc time

Description

Add a new chunk of special memory to the specified pool.

Returns 0 on success or a -ve errno on failure.

A call to gen_pool_add() will place the size bytes of memory starting at addr (in the kernel's virtual address space) into the given pool, once again using nid as the node ID for ancillary memory allocations. The gen_pool_add_virt() variant associates an explicit physical address with the memory; this is only necessary if the pool will be used for DMA allocations.

The functions for allocating memory from the pool (and putting it back) are:

unsigned long **gen_pool_alloc**(struct gen_pool *pool, size_t size)

allocate special memory from the pool

Parameters

struct gen_pool *pool
pool to allocate from

size_t size
number of bytes to allocate from the pool

Description

Allocate the requested number of bytes from the specified pool. Uses the pool allocation function (with first-fit algorithm by default). Can not be used in NMI handler on architectures without NMI-safe cmpxchg implementation.

void ***gen_pool_dma_alloc**(struct gen_pool *pool, size_t size, dma_addr_t *dma)

allocate special memory from the pool for DMA usage

Parameters

struct gen_pool *pool
pool to allocate from

size_t size
number of bytes to allocate from the pool

dma_addr_t *dma
dma-view physical address return value. Use `NULL` if unneeded.

Description

Allocate the requested number of bytes from the specified pool. Uses the pool allocation function (with first-fit algorithm by default). Can not be used in NMI handler on architectures without NMI-safe cmpxchg implementation.

Return

virtual address of the allocated memory, or `NULL` on failure

void **gen_pool_free_owner**(struct gen_pool *pool, unsigned long addr, size_t size, void **owner)

free allocated special memory back to the pool

Parameters

struct gen_pool *pool
pool to free to

unsigned long addr
starting address of memory to free back to pool

size_t size
size in bytes of memory to free

void **owner
private data stashed at gen_pool_add() time

Description

Free previously allocated special memory back to the specified pool. Can not be used in NMI handler on architectures without NMI-safe cmpxchg implementation.

As one would expect, gen_pool_alloc() will allocate size< bytes from the given pool. The gen_pool_dma_alloc() variant allocates memory for use with DMA operations, returning the associated physical address in the space pointed to by dma. This will only work if the memory was added with gen_pool_add_virt(). Note that this function departs from the usual genpool pattern of using unsigned long values to represent kernel addresses; it returns a void * instead.

That all seems relatively simple; indeed, some developers clearly found it to be too simple. After all, the interface above provides no control over how the allocation functions choose which specific piece of memory to return. If that sort of control is needed, the following functions will be of interest:

unsigned long **gen_pool_alloc_algo_owner**(struct gen_pool *pool, size_t size, genpool_algo_t algo, void *data, void **owner)

allocate special memory from the pool

Parameters

struct gen_pool *pool
pool to allocate from

size_t size
number of bytes to allocate from the pool

genpool_algo_t algo
algorithm passed from caller

void *data
data passed to algorithm

void **owner
optionally retrieve the chunk owner

Description

Allocate the requested number of bytes from the specified pool. Uses the pool allocation function (with first-fit algorithm by default). Can not be used in NMI handler on architectures without NMI-safe cmpxchg implementation.

void **gen_pool_set_algo**(struct gen_pool *pool, genpool_algo_t algo, void *data)

set the allocation algorithm

Parameters

struct gen_pool *pool
pool to change allocation algorithm

genpool_algo_t algo
custom algorithm function

void *data
additional data used by **algo**

Description

Call **algo** for each memory allocation in the pool. If **algo** is NULL use gen_pool_first_fit as default memory allocation function.

Allocations with gen_pool_alloc_algo() specify an algorithm to be used to choose the memory to be allocated; the default algorithm can be set with gen_pool_set_algo(). The data value is passed to the algorithm; most ignore it, but it is occasionally needed. One can, naturally, write a special-purpose algorithm, but there is a fair set already available:

- gen_pool_first_fit is a simple first-fit allocator; this is the default algorithm if none other has been specified.
- gen_pool_first_fit_align forces the allocation to have a specific alignment (passed via data in a genpool_data_align structure).
- gen_pool_first_fit_order_align aligns the allocation to the order of the size. A 60-byte allocation will thus be 64-byte aligned, for example.
- gen_pool_best_fit, as one would expect, is a simple best-fit allocator.
- gen_pool_fixed_alloc allocates at a specific offset (passed in a genpool_data_fixed structure via the data parameter) within the pool. If the indicated memory is not available the allocation fails.

There is a handful of other functions, mostly for purposes like querying the space available in the pool or iterating through chunks of memory. Most users, however, should not need much beyond what has been described above. With luck, wider awareness of this module will help to prevent the writing of special-purpose memory allocators in the future.

phys_addr_t **gen_pool_virt_to_phys**(struct gen_pool *pool, unsigned long addr)
return the physical address of memory

Parameters

struct gen_pool *pool
pool to allocate from

unsigned long addr
starting address of memory

Description

Returns the physical address on success, or -1 on error.

void **gen_pool_for_each_chunk**(struct gen_pool *pool, void (*func)(struct gen_pool *pool, struct gen_pool_chunk *chunk, void *data), void *data)
call func for every chunk of generic memory pool

Parameters

struct gen_pool *pool
the generic memory pool

void (*func)(struct gen_pool *pool, struct gen_pool_chunk *chunk, void *data)
func to call

void *data
  additional data used by **func**

Description

Call **func** for every chunk of generic memory pool. The **func** is called with rcu_read_lock held.

bool **gen_pool_has_addr**(struct gen_pool *pool, unsigned long start, size_t size)
  checks if an address falls within the range of a pool

Parameters

struct gen_pool *pool
  the generic memory pool

unsigned long start
  start address

size_t size
  size of the region

Description

Check if the range of addresses falls within the specified pool. Returns true if the entire range is contained in the pool and false otherwise.

size_t **gen_pool_avail**(struct gen_pool *pool)
  get available free space of the pool

Parameters

struct gen_pool *pool
  pool to get available free space

Description

Return available free space of the specified pool.

size_t **gen_pool_size**(struct gen_pool *pool)
  get size in bytes of memory managed by the pool

Parameters

struct gen_pool *pool
  pool to get size

Description

Return size in bytes of memory managed by the pool.

struct gen_pool ***gen_pool_get**(struct device *dev, const char *name)
  Obtain the gen_pool (if any) for a device

Parameters

struct device *dev
  device to retrieve the gen_pool from

const char *name
  name of a gen_pool or NULL, identifies a particular gen_pool on device

Description

Returns the gen_pool for the device if one is present, or NULL.

struct gen_pool ***of_gen_pool_get**(struct device_node *np, const char *propname, int index)

find a pool by phandle property

Parameters

struct device_node *np
device node

const char *propname
property name containing phandle(s)

int index
index into the phandle array

Description

Returns the pool that contains the chunk starting at the physical address of the device tree node pointed at by the phandle property, or NULL if not found.

6.9 pin_user_pages() and related calls

6.9.1 Overview

This document describes the following functions:

```
pin_user_pages()
pin_user_pages_fast()
pin_user_pages_remote()
```

6.9.2 Basic description of FOLL_PIN

FOLL_PIN and FOLL_LONGTERM are flags that can be passed to the get_user_pages*() ("gup") family of functions. FOLL_PIN has significant interactions and interdependencies with FOLL_LONGTERM, so both are covered here.

FOLL_PIN is internal to gup, meaning that it should not appear at the gup call sites. This allows the associated wrapper functions (pin_user_pages*() and others) to set the correct combination of these flags, and to check for problems as well.

FOLL_LONGTERM, on the other hand, *is* allowed to be set at the gup call sites. This is in order to avoid creating a large number of wrapper functions to cover all combinations of get*(), pin*(), FOLL_LONGTERM, and more. Also, the pin_user_pages*() APIs are clearly distinct from the get_user_pages*() APIs, so that's a natural dividing line, and a good point to make separate wrapper calls. In other words, use pin_user_pages*() for DMA-pinned pages, and get_user_pages*() for other cases. There are five cases described later on in this document, to further clarify that concept.

FOLL_PIN and FOLL_GET are mutually exclusive for a given gup call. However, multiple threads and call sites are free to pin the same struct pages, via both FOLL_PIN and FOLL_GET. It's just the call site that needs to choose one or the other, not the struct page(s).

The FOLL_PIN implementation is nearly the same as FOLL_GET, except that FOLL_PIN uses a different reference counting technique.

FOLL_PIN is a prerequisite to FOLL_LONGTERM. Another way of saying that is, FOLL_LONGTERM is a specific case, more restrictive case of FOLL_PIN.

6.9.3 Which flags are set by each wrapper

For these pin_user_pages*() functions, FOLL_PIN is OR'd in with whatever gup flags the caller provides. The caller is required to pass in a non-null struct pages* array, and the function then pins pages by incrementing each by a special value: GUP_PIN_COUNTING_BIAS.

For large folios, the GUP_PIN_COUNTING_BIAS scheme is not used. Instead, the extra space available in the `struct folio` is used to store the pincount directly.

This approach for large folios avoids the counting upper limit problems that are discussed below. Those limitations would have been aggravated severely by huge pages, because each tail page adds a refcount to the head page. And in fact, testing revealed that, without a separate pincount field, refcount overflows were seen in some huge page stress tests.

This also means that huge pages and large folios do not suffer from the false positives problem that is mentioned below.:

```
Function
--------
pin_user_pages          FOLL_PIN is always set internally by this function.
pin_user_pages_fast     FOLL_PIN is always set internally by this function.
pin_user_pages_remote   FOLL_PIN is always set internally by this function.
```

For these get_user_pages*() functions, FOLL_GET might not even be specified. Behavior is a little more complex than above. If FOLL_GET was *not* specified, but the caller passed in a non-null struct pages* array, then the function sets FOLL_GET for you, and proceeds to pin pages by incrementing the refcount of each page by +1.:

```
Function
--------
get_user_pages           FOLL_GET is sometimes set internally by this function.
get_user_pages_fast      FOLL_GET is sometimes set internally by this function.
get_user_pages_remote    FOLL_GET is sometimes set internally by this function.
```

6.9.4 Tracking dma-pinned pages

Some of the key design constraints, and solutions, for tracking dma-pinned pages:

- An actual reference count, per struct page, is required. This is because multiple processes may pin and unpin a page.
- False positives (reporting that a page is dma-pinned, when in fact it is not) are acceptable, but false negatives are not.
- struct page may not be increased in size for this, and all fields are already used.
- Given the above, we can overload the page->_refcount field by using, sort of, the upper bits in that field for a dma-pinned count. "Sort of", means that, rather than dividing page->_refcount into bit fields, we simple add a medium- large value (GUP_PIN_COUNTING_BIAS, initially chosen to be 1024: 10 bits) to page->_refcount. This provides fuzzy behavior: if a page has get_page() called on it 1024 times, then it will appear to have a single dma-pinned count. And again, that's acceptable.

This also leads to limitations: there are only 31-10==21 bits available for a counter that increments 10 bits at a time.

- Because of that limitation, special handling is applied to the zero pages when using FOLL_PIN. We only pretend to pin a zero page - we don't alter its refcount or pincount at all (it is permanent, so there's no need). The unpinning functions also don't do anything to a zero page. This is transparent to the caller.
- Callers must specifically request "dma-pinned tracking of pages". In other words, just calling get_user_pages() will not suffice; a new set of functions, pin_user_page() and related, must be used.

6.9.5 FOLL_PIN, FOLL_GET, FOLL_LONGTERM: when to use which flags

Thanks to Jan Kara, Vlastimil Babka and several other -mm people, for describing these categories:

CASE 1: Direct IO (DIO)

There are GUP references to pages that are serving as DIO buffers. These buffers are needed for a relatively short time (so they are not "long term"). No special synchronization with page_mkclean() or munmap() is provided. Therefore, flags to set at the call site are:

```
FOLL_PIN
```

…but rather than setting FOLL_PIN directly, call sites should use one of the pin_user_pages*() routines that set FOLL_PIN.

CASE 2: RDMA

There are GUP references to pages that are serving as DMA buffers. These buffers are needed for a long time ("long term"). No special synchronization with page_mkclean() or munmap() is provided. Therefore, flags to set at the call site are:

```
FOLL_PIN | FOLL_LONGTERM
```

NOTE: Some pages, such as DAX pages, cannot be pinned with longterm pins. That's because DAX pages do not have a separate page cache, and so "pinning" implies locking down file system blocks, which is not (yet) supported in that way.

CASE 3: MMU notifier registration, with or without page faulting hardware

Device drivers can pin pages via get_user_pages*(), and register for mmu notifier callbacks for the memory range. Then, upon receiving a notifier "invalidate range" callback , stop the device from using the range, and unpin the pages. There may be other possible schemes, such as for example explicitly synchronizing against pending IO, that accomplish approximately the same thing.

Or, if the hardware supports replayable page faults, then the device driver can avoid pinning entirely (this is ideal), as follows: register for mmu notifier callbacks as above, but instead of stopping the device and unpinning in the callback, simply remove the range from the device's page tables.

Either way, as long as the driver unpins the pages upon mmu notifier callback, then there is proper synchronization with both filesystem and mm (page_mkclean(), munmap(), etc). Therefore, neither flag needs to be set.

CASE 4: Pinning for struct page manipulation only

If only struct page data (as opposed to the actual memory contents that a page is tracking) is affected, then normal GUP calls are sufficient, and neither flag needs to be set.

CASE 5: Pinning in order to write to the data within the page

Even though neither DMA nor Direct IO is involved, just a simple case of "pin, write to a page's data, unpin" can cause a problem. Case 5 may be considered a superset of Case 1, plus Case 2, plus anything that invokes that pattern. In other words, if the code is neither Case 1 nor Case 2, it may still require FOLL_PIN, for patterns like this:

Correct (uses FOLL_PIN calls):
pin_user_pages() write to the data within the pages unpin_user_pages()

INCORRECT (uses FOLL_GET calls):
get_user_pages() write to the data within the pages put_page()

6.9.6 page_maybe_dma_pinned(): the whole point of pinning

The whole point of marking pages as "DMA-pinned" or "gup-pinned" is to be able to query, "is this page DMA-pinned?" That allows code such as page_mkclean() (and file system writeback code in general) to make informed decisions about what to do when a page cannot be unmapped due to such pins.

What to do in those cases is the subject of a years-long series of discussions and debates (see the References at the end of this document). It's a TODO item here: fill in the details once that's worked out. Meanwhile, it's safe to say that having this available:

```
static inline bool page_maybe_dma_pinned(struct page *page)
```

...is a prerequisite to solving the long-running gup+DMA problem.

6.9.7 Another way of thinking about FOLL_GET, FOLL_PIN, and FOLL_LONGTERM

Another way of thinking about these flags is as a progression of restrictions: FOLL_GET is for struct page manipulation, without affecting the data that the struct page refers to. FOLL_PIN is a *replacement* for FOLL_GET, and is for short term pins on pages whose data *will* get accessed. As such, FOLL_PIN is a "more severe" form of pinning. And finally, FOLL_LONGTERM is an even more restrictive case that has FOLL_PIN as a prerequisite: this is for pages that will be pinned longterm, and whose data will be accessed.

6.9.8 Unit testing

This file:

```
tools/testing/selftests/mm/gup_test.c
```

has the following new calls to exercise the new pin*() wrapper functions:

- PIN_FAST_BENCHMARK (./gup_test -a)
- PIN_BASIC_TEST (./gup_test -b)

You can monitor how many total dma-pinned pages have been acquired and released since the system was booted, via two new /proc/vmstat entries:

```
/proc/vmstat/nr_foll_pin_acquired
/proc/vmstat/nr_foll_pin_released
```

Under normal conditions, these two values will be equal unless there are any long-term [R]DMA pins in place, or during pin/unpin transitions.

- nr_foll_pin_acquired: This is the number of logical pins that have been acquired since the system was powered on. For huge pages, the head page is pinned once for each page (head page and each tail page) within the huge page. This follows the same sort of behavior that get_user_pages() uses for huge pages: the head page is refcounted once for each tail or head page in the huge page, when get_user_pages() is applied to a huge page.
- nr_foll_pin_released: The number of logical pins that have been released since the system was powered on. Note that pages are released (unpinned) on a PAGE_SIZE granularity, even if the original pin was applied to a huge page. Becaused of the pin count behavior described above in "nr_foll_pin_acquired", the accounting balances out, so that after doing this:

  ```
  pin_user_pages(huge_page);
  for (each page in huge_page)
      unpin_user_page(page);
  ```

…the following is expected:

```
nr_foll_pin_released == nr_foll_pin_acquired
```

(…unless it was already out of balance due to a long-term RDMA pin being in place.)

6.9.9 Other diagnostics

dump_page() has been enhanced slightly to handle these new counting fields, and to better report on large folios in general. Specifically, for large folios, the exact pincount is reported.

6.9.10 References

- Some slow progress on get_user_pages() (Apr 2, 2019)
- DMA and get_user_pages() (LPC: Dec 12, 2018)
- The trouble with get_user_pages() (Apr 30, 2018)
- LWN kernel index: get_user_pages()

John Hubbard, October, 2019

6.10 Boot time memory management

Early system initialization cannot use "normal" memory management simply because it is not set up yet. But there is still need to allocate memory for various data structures, for instance for the physical page allocator.

A specialized allocator called `memblock` performs the boot time memory management. The architecture specific initialization must set it up in `setup_arch()` and tear it down in `mem_init()` functions.

Once the early memory management is available it offers a variety of functions and macros for memory allocations. The allocation request may be directed to the first (and probably the only) node or to a particular node in a NUMA system. There are API variants that panic when an allocation fails and those that don't.

Memblock also offers a variety of APIs that control its own behaviour.

6.10.1 Memblock Overview

Memblock is a method of managing memory regions during the early boot period when the usual kernel memory allocators are not up and running.

Memblock views the system memory as collections of contiguous regions. There are several types of these collections:

- `memory` - describes the physical memory available to the kernel; this may differ from the actual physical memory installed in the system, for instance when the memory is restricted with `mem=` command line parameter
- `reserved` - describes the regions that were allocated
- `physmem` - describes the actual physical memory available during boot regardless of the possible restrictions and memory hot(un)plug; the `physmem` type is only available on some architectures.

Each region is represented by *`struct memblock_region`* that defines the region extents, its attributes and NUMA node id on NUMA systems. Every memory type is described by the *`struct memblock_type`* which contains an array of memory regions along with the allocator metadata. The "memory" and "reserved" types are nicely wrapped with *`struct memblock`*. This structure is statically initialized at build time. The region arrays are initially sized to `INIT_MEMBLOCK_MEMORY_REGIONS` for "memory" and `INIT_MEMBLOCK_RESERVED_REGIONS` for "reserved". The region array for "physmem" is initially sized to `INIT_PHYSMEM_REGIONS`. The memblock_allow_resize() enables automatic resizing of the region arrays during addition of new regions. This feature should be used with care so that memory allocated for the region array will not overlap with areas that should be reserved, for example initrd.

The early architecture setup should tell memblock what the physical memory layout is by using *memblock_add()* or *memblock_add_node()* functions. The first function does not assign the region to a NUMA node and it is appropriate for UMA systems. Yet, it is possible to use it on NUMA systems as well and assign the region to a NUMA node later in the setup process using *memblock_set_node()*. The *memblock_add_node()* performs such an assignment directly.

Once memblock is setup the memory can be allocated using one of the API variants:

- memblock_phys_alloc*() - these functions return the **physical** address of the allocated memory
- memblock_alloc*() - these functions return the **virtual** address of the allocated memory.

Note, that both API variants use implicit assumptions about allowed memory ranges and the fallback methods. Consult the documentation of *memblock_alloc_internal()* and *memblock_alloc_range_nid()* functions for more elaborate description.

As the system boot progresses, the architecture specific mem_init() function frees all the memory to the buddy page allocator.

Unless an architecture enables `CONFIG_ARCH_KEEP_MEMBLOCK`, the memblock data structures (except "physmem") will be discarded after the system initialization completes.

6.10.2 Functions and structures

Here is the description of memblock data structures, functions and macros. Some of them are actually internal, but since they are documented it would be silly to omit them. Besides, reading the descriptions for the internal functions can help to understand what really happens under the hood.

enum **memblock_flags**

definition of memory region attributes

Constants

MEMBLOCK_NONE
no special request

MEMBLOCK_HOTPLUG
memory region indicated in the firmware-provided memory map during early boot as hot(un)pluggable system RAM (e.g., memory range that might get hotunplugged later). With "movable_node" set on the kernel commandline, try keeping this memory region hotunpluggable. Does not apply to memblocks added ("hotplugged") after early boot.

MEMBLOCK_MIRROR
mirrored region

MEMBLOCK_NOMAP
don't add to kernel direct mapping and treat as reserved in the memory map; refer to *memblock_mark_nomap()* description for further details

MEMBLOCK_DRIVER_MANAGED
memory region that is always detected and added via a driver, and never indicated

in the firmware-provided memory map as system RAM. This corresponds to IORESOURCE_SYSRAM_DRIVER_MANAGED in the kernel resource tree.

struct **memblock_region**

represents a memory region

Definition:

```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    enum memblock_flags flags;
#ifdef CONFIG_NUMA;
    int nid;
#endif;
};
```

Members

base
base address of the region

size
size of the region

flags
memory region attributes

nid
NUMA node id

struct **memblock_type**

collection of memory regions of certain type

Definition:

```
struct memblock_type {
    unsigned long cnt;
    unsigned long max;
    phys_addr_t total_size;
    struct memblock_region *regions;
    char *name;
};
```

Members

cnt
number of regions

max
size of the allocated array

total_size
size of all regions

regions
array of regions

name
the memory type symbolic name

struct **memblock**
memblock allocator metadata

Definition:

```
struct memblock {
    bool bottom_up;
    phys_addr_t current_limit;
    struct memblock_type memory;
    struct memblock_type reserved;
};
```

Members

bottom_up
is bottom up direction?

current_limit
physical address of the current allocation limit

memory
usable memory regions

reserved
reserved memory regions

for_each_physmem_range

`for_each_physmem_range (i, type, p_start, p_end)`

iterate through physmem areas not included in type.

Parameters

i
u64 used as loop variable

type
ptr to memblock_type which excludes from the iteration, can be `NULL`

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

__for_each_mem_range

`__for_each_mem_range (i, type_a, type_b, nid, flags, p_start, p_end, p_nid)`

iterate through memblock areas from type_a and not included in type_b. Or just type_a if type_b is NULL.

Parameters

i
u64 used as loop variable

type_a
ptr to memblock_type to iterate

type_b
ptr to memblock_type which excludes from the iteration

nid
node selector, `NUMA_NO_NODE` for all nodes

flags
pick from blocks based on memory attributes

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

p_nid
ptr to int for nid of the range, can be `NULL`

__for_each_mem_range_rev

`__for_each_mem_range_rev (i, type_a, type_b, nid, flags, p_start, p_end, p_nid)`

reverse iterate through memblock areas from type_a and not included in type_b. Or just type_a if type_b is NULL.

Parameters

i
u64 used as loop variable

type_a
ptr to memblock_type to iterate

type_b
ptr to memblock_type which excludes from the iteration

nid
node selector, `NUMA_NO_NODE` for all nodes

flags
pick from blocks based on memory attributes

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

p_nid
ptr to int for nid of the range, can be `NULL`

for_each_mem_range

`for_each_mem_range (i, p_start, p_end)`

iterate through memory areas.

Parameters

i
u64 used as loop variable

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

for_each_mem_range_rev

`for_each_mem_range_rev (i, p_start, p_end)`

reverse iterate through memblock areas from type_a and not included in type_b. Or just type_a if type_b is NULL.

Parameters

i
u64 used as loop variable

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

for_each_reserved_mem_range

`for_each_reserved_mem_range (i, p_start, p_end)`

iterate over all reserved memblock areas

Parameters

i
u64 used as loop variable

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

Description

Walks over reserved areas of memblock. Available as soon as memblock is initialized.

for_each_mem_pfn_range

`for_each_mem_pfn_range (i, nid, p_start, p_end, p_nid)`

early memory pfn range iterator

Parameters

i
an integer used as loop variable

nid
node selector, `MAX_NUMNODES` for all nodes

p_start
ptr to ulong for start pfn of the range, can be NULL

p_end
ptr to ulong for end pfn of the range, can be NULL

p_nid
ptr to int for nid of the range, can be NULL

Description

Walks over configured memory ranges.

for_each_free_mem_pfn_range_in_zone

`for_each_free_mem_pfn_range_in_zone (i, zone, p_start, p_end)`

iterate through zone specific free memblock areas

Parameters

i
u64 used as loop variable

zone
zone in which all of the memory blocks reside

p_start
ptr to phys_addr_t for start address of the range, can be NULL

p_end
ptr to phys_addr_t for end address of the range, can be NULL

Description

Walks over free (memory && !reserved) areas of memblock in a specific zone. Available once memblock and an empty zone is initialized. The main assumption is that the zone start, end, and pgdat have been associated. This way we can use the zone to determine NUMA node, and if a given part of the memblock is valid for the zone.

for_each_free_mem_pfn_range_in_zone_from

`for_each_free_mem_pfn_range_in_zone_from (i, zone, p_start, p_end)`

iterate through zone specific free memblock areas from a given point

Parameters

i
u64 used as loop variable

zone
zone in which all of the memory blocks reside

p_start
ptr to phys_addr_t for start address of the range, can be NULL

p_end
ptr to phys_addr_t for end address of the range, can be NULL

Description

Walks over free (memory && !reserved) areas of memblock in a specific zone, continuing from current position. Available as soon as memblock is initialized.

for_each_free_mem_range

`for_each_free_mem_range (i, nid, flags, p_start, p_end, p_nid)`

iterate through free memblock areas

Parameters

i
u64 used as loop variable

nid
node selector, `NUMA_NO_NODE` for all nodes

flags
pick from blocks based on memory attributes

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

p_nid
ptr to int for nid of the range, can be `NULL`

Description

Walks over free (memory && !reserved) areas of memblock. Available as soon as memblock is initialized.

for_each_free_mem_range_reverse

`for_each_free_mem_range_reverse (i, nid, flags, p_start, p_end, p_nid)`

rev-iterate through free memblock areas

Parameters

i
u64 used as loop variable

nid
node selector, `NUMA_NO_NODE` for all nodes

flags
pick from blocks based on memory attributes

p_start
ptr to phys_addr_t for start address of the range, can be `NULL`

p_end
ptr to phys_addr_t for end address of the range, can be `NULL`

p_nid
ptr to int for nid of the range, can be `NULL`

Description

Walks over free (memory && !reserved) areas of memblock in reverse order. Available as soon as memblock is initialized.

void **memblock_set_current_limit**(phys_addr_t limit)

Set the current allocation limit to allow limiting allocations to what is currently accessible during boot

Parameters

phys_addr_t limit
New limit value (physical address)

unsigned long **memblock_region_memory_base_pfn**(const struct *memblock_region* *reg)

get the lowest pfn of the memory region

Parameters

const struct memblock_region *reg
memblock_region structure

Return

the lowest pfn intersecting with the memory region

unsigned long **memblock_region_memory_end_pfn**(const struct *memblock_region* *reg)

get the end pfn of the memory region

Parameters

const struct memblock_region *reg
memblock_region structure

Return

the end_pfn of the reserved region

unsigned long **memblock_region_reserved_base_pfn**(const struct *memblock_region* *reg)

get the lowest pfn of the reserved region

Parameters

const struct memblock_region *reg
memblock_region structure

Return

the lowest pfn intersecting with the reserved region

unsigned long **memblock_region_reserved_end_pfn**(const struct *memblock_region* *reg)

get the end pfn of the reserved region

Parameters

const struct memblock_region *reg
memblock_region structure

Return

the end_pfn of the reserved region

for_each_mem_region

`for_each_mem_region (region)`

itereate over memory regions

Parameters

region
loop variable

for_each_reserved_mem_region

`for_each_reserved_mem_region (region)`

itereate over reserved memory regions

Parameters

region
loop variable

phys_addr_t __init_memblock **__memblock_find_range_bottom_up**(phys_addr_t start, phys_addr_t end, phys_addr_t size, phys_addr_t align, int nid, enum *memblock_flags* flags)

find free area utility in bottom-up

Parameters

phys_addr_t start
start of candidate range

phys_addr_t end
end of candidate range, can be `MEMBLOCK_ALLOC_ANYWHERE` or `MEMBLOCK_ALLOC_ACCESSIBLE`

phys_addr_t size
size of free area to find

phys_addr_t align
alignment of free area to find

int nid
nid of the free area to find, `NUMA_NO_NODE` for any node

enum memblock_flags flags
pick from blocks based on memory attributes

Description

Utility called from *memblock_find_in_range_node()*, find free area bottom-up.

Return

Found address on success, 0 on failure.

phys_addr_t __init_memblock **__memblock_find_range_top_down**(phys_addr_t start, phys_addr_t end, phys_addr_t size, phys_addr_t align, int nid, enum *memblock_flags* flags)

find free area utility, in top-down

Parameters

phys_addr_t start
start of candidate range

phys_addr_t end
end of candidate range, can be MEMBLOCK_ALLOC_ANYWHERE or MEMBLOCK_ALLOC_ACCESSIBLE

phys_addr_t size
size of free area to find

phys_addr_t align
alignment of free area to find

int nid
nid of the free area to find, NUMA_NO_NODE for any node

enum memblock_flags flags
pick from blocks based on memory attributes

Description

Utility called from *memblock_find_in_range_node()*, find free area top-down.

Return

Found address on success, 0 on failure.

phys_addr_t __init_memblock **memblock_find_in_range_node**(phys_addr_t size, phys_addr_t align, phys_addr_t start, phys_addr_t end, int nid, enum *memblock_flags* flags)

find free area in given range and node

Parameters

phys_addr_t size
size of free area to find

phys_addr_t align
alignment of free area to find

phys_addr_t start
start of candidate range

phys_addr_t end
end of candidate range, can be MEMBLOCK_ALLOC_ANYWHERE or MEMBLOCK_ALLOC_ACCESSIBLE

int nid
nid of the free area to find, NUMA_NO_NODE for any node

enum memblock_flags flags
pick from blocks based on memory attributes

Description

Find **size** free area aligned to **align** in the specified range and node.

Return

Found address on success, 0 on failure.

phys_addr_t __init_memblock **memblock_find_in_range**(phys_addr_t start, phys_addr_t end, phys_addr_t size, phys_addr_t align)
find free area in given range

Parameters

phys_addr_t start
start of candidate range

phys_addr_t end
end of candidate range, can be `MEMBLOCK_ALLOC_ANYWHERE` or `MEMBLOCK_ALLOC_ACCESSIBLE`

phys_addr_t size
size of free area to find

phys_addr_t align
alignment of free area to find

Description

Find **size** free area aligned to **align** in the specified range.

Return

Found address on success, 0 on failure.

void **memblock_discard**(void)
discard memory and reserved arrays if they were allocated

Parameters

void
no arguments

int __init_memblock **memblock_double_array**(struct *memblock_type* *type, phys_addr_t new_area_start, phys_addr_t new_area_size)
double the size of the memblock regions array

Parameters

struct memblock_type *type
memblock type of the regions array being doubled

phys_addr_t new_area_start
starting address of memory range to avoid overlap with

phys_addr_t new_area_size
size of memory range to avoid overlap with

Description

Double the size of the **type** regions array. If memblock is being used to allocate memory for a new reserved regions array and there is a previously allocated memory range [**new_area_start**, **new_area_start** + **new_area_size**] waiting to be reserved, ensure the memory used by the new array does not overlap.

Return

0 on success, -1 on failure.

void __init_memblock **memblock_merge_regions**(struct *memblock_type* *type, unsigned long start_rgn, unsigned long end_rgn)

merge neighboring compatible regions

Parameters

struct memblock_type *type
memblock type to scan

unsigned long start_rgn
start scanning from (**start_rgn** - 1)

unsigned long end_rgn
end scanning at (**end_rgn** - 1) Scan **type** and merge neighboring compatible regions in [**start_rgn** - 1, **end_rgn**)

void __init_memblock **memblock_insert_region**(struct *memblock_type* *type, int idx, phys_addr_t base, phys_addr_t size, int nid, enum *memblock_flags* flags)

insert new memblock region

Parameters

struct memblock_type *type
memblock type to insert into

int idx
index for the insertion point

phys_addr_t base
base address of the new region

phys_addr_t size
size of the new region

int nid
node id of the new region

enum memblock_flags flags
flags of the new region

Description

Insert new memblock region [**base**, **base** + **size**) into **type** at **idx**. **type** must already have extra room to accommodate the new region.

int __init_memblock **memblock_add_range**(struct *memblock_type* *type, phys_addr_t base, phys_addr_t size, int nid, enum *memblock_flags* flags) add new memblock region

Parameters

struct memblock_type *type
memblock type to add new region into

phys_addr_t base
base address of the new region

phys_addr_t size
size of the new region

int nid
nid of the new region

enum memblock_flags flags
flags of the new region

Description

Add new memblock region [**base**, **base** + **size**) into **type**. The new region is allowed to overlap with existing ones - overlaps don't affect already existing regions. **type** is guaranteed to be minimal (all neighbouring compatible regions are merged) after the addition.

Return

0 on success, -errno on failure.

int __init_memblock **memblock_add_node**(phys_addr_t base, phys_addr_t size, int nid, enum *memblock_flags* flags)

add new memblock region within a NUMA node

Parameters

phys_addr_t base
base address of the new region

phys_addr_t size
size of the new region

int nid
nid of the new region

enum memblock_flags flags
flags of the new region

Description

Add new memblock region [**base**, **base** + **size**) to the "memory" type. See *memblock_add_range()* description for mode details

Return

0 on success, -errno on failure.

int __init_memblock **memblock_add**(phys_addr_t base, phys_addr_t size)

add new memblock region

Parameters

phys_addr_t base
base address of the new region

phys_addr_t size
size of the new region

Description

Add new memblock region [**base**, **base** + **size**) to the "memory" type. See *memblock_add_range()* description for mode details

Return

0 on success, -errno on failure.

int __init_memblock **memblock_isolate_range**(struct *memblock_type* *type, phys_addr_t base, phys_addr_t size, int *start_rgn, int *end_rgn)

isolate given range into disjoint memblocks

Parameters

struct memblock_type *type
memblock type to isolate range for

phys_addr_t base
base of range to isolate

phys_addr_t size
size of range to isolate

int *start_rgn
out parameter for the start of isolated region

int *end_rgn
out parameter for the end of isolated region

Description

Walk **type** and ensure that regions don't cross the boundaries defined by [**base**, **base** + **size**). Crossing regions are split at the boundaries, which may create at most two more regions. The index of the first region inside the range is returned in ***start_rgn** and end in ***end_rgn**.

Return

0 on success, -errno on failure.

void __init_memblock **memblock_free**(void *ptr, size_t size)

free boot memory allocation

Parameters

void *ptr
starting address of the boot memory allocation

size_t size
size of the boot memory block in bytes

Description

Free boot memory block previously allocated by memblock_alloc_xx() API. The freeing memory will not be released to the buddy allocator.

int __init_memblock **memblock_phys_free**(phys_addr_t base, phys_addr_t size)
free boot memory block

Parameters

phys_addr_t base
phys starting address of the boot memory block

phys_addr_t size
size of the boot memory block in bytes

Description

Free boot memory block previously allocated by memblock_phys_alloc_xx() API. The freeing memory will not be released to the buddy allocator.

int __init_memblock **memblock_setclr_flag**(phys_addr_t base, phys_addr_t size, int set, int flag)
set or clear flag for a memory region

Parameters

phys_addr_t base
base address of the region

phys_addr_t size
size of the region

int set
set or clear the flag

int flag
the flag to update

Description

This function isolates region [**base**, **base** + **size**), and sets/clears flag

Return

0 on success, -errno on failure.

int __init_memblock **memblock_mark_hotplug**(phys_addr_t base, phys_addr_t size)
Mark hotpluggable memory with flag MEMBLOCK_HOTPLUG.

Parameters

phys_addr_t base
the base phys addr of the region

phys_addr_t size
the size of the region

Return

0 on success, -errno on failure.

int __init_memblock **memblock_clear_hotplug**(phys_addr_t base, phys_addr_t size)
Clear flag MEMBLOCK_HOTPLUG for a specified region.

Parameters

phys_addr_t base
the base phys addr of the region

phys_addr_t size
the size of the region

Return

0 on success, -errno on failure.

int __init_memblock **memblock_mark_mirror**(phys_addr_t base, phys_addr_t size)
Mark mirrored memory with flag MEMBLOCK_MIRROR.

Parameters

phys_addr_t base
the base phys addr of the region

phys_addr_t size
the size of the region

Return

0 on success, -errno on failure.

int __init_memblock **memblock_mark_nomap**(phys_addr_t base, phys_addr_t size)
Mark a memory region with flag MEMBLOCK_NOMAP.

Parameters

phys_addr_t base
the base phys addr of the region

phys_addr_t size
the size of the region

Description

The memory regions marked with `MEMBLOCK_NOMAP` will not be added to the direct mapping of the physical memory. These regions will still be covered by the memory map. The struct page representing NOMAP memory frames in the memory map will be PageReserved()

Note

if the memory being marked `MEMBLOCK_NOMAP` was allocated from memblock, the caller must inform kmemleak to ignore that memory

Return

0 on success, -errno on failure.

int __init_memblock **memblock_clear_nomap**(phys_addr_t base, phys_addr_t size)
Clear flag MEMBLOCK_NOMAP for a specified region.

Parameters

phys_addr_t base
the base phys addr of the region

phys_addr_t size
the size of the region

Return

0 on success, -errno on failure.

void **__next_mem_range**(u64 *idx, int nid, enum *memblock_flags* flags, struct *memblock_type* *type_a, struct *memblock_type* *type_b, phys_addr_t *out_start, phys_addr_t *out_end, int *out_nid)

next function for *for_each_free_mem_range()* etc.

Parameters

u64 *idx
pointer to u64 loop variable

int nid
node selector, `NUMA_NO_NODE` for all nodes

enum memblock_flags flags
pick from blocks based on memory attributes

struct memblock_type *type_a
pointer to memblock_type from where the range is taken

struct memblock_type *type_b
pointer to memblock_type which excludes memory from being taken

phys_addr_t *out_start
ptr to phys_addr_t for start address of the range, can be `NULL`

phys_addr_t *out_end
ptr to phys_addr_t for end address of the range, can be `NULL`

int *out_nid
ptr to int for nid of the range, can be `NULL`

Description

Find the first area from ***idx** which matches **nid**, fill the out parameters, and update ***idx** for the next iteration. The lower 32bit of ***idx** contains index into type_a and the upper 32bit indexes the areas before each region in type_b. For example, if type_b regions look like the following,

0:[0-16), 1:[32-48), 2:[128-130)

The upper 32bit indexes the following regions.

0:[0-0), 1:[16-32), 2:[48-128), 3:[130-MAX)

As both region arrays are sorted, the function advances the two indices in lockstep and returns each intersection.

void __init_memblock **__next_mem_range_rev**(u64 *idx, int nid, enum *memblock_flags* flags, struct *memblock_type* *type_a, struct *memblock_type* *type_b, phys_addr_t *out_start, phys_addr_t *out_end, int *out_nid)

generic next function for for_each_*_range_rev()

Parameters

u64 *idx
pointer to u64 loop variable

int nid
node selector, NUMA_NO_NODE for all nodes

enum memblock_flags flags
pick from blocks based on memory attributes

struct memblock_type *type_a
pointer to memblock_type from where the range is taken

struct memblock_type *type_b
pointer to memblock_type which excludes memory from being taken

phys_addr_t *out_start
ptr to phys_addr_t for start address of the range, can be NULL

phys_addr_t *out_end
ptr to phys_addr_t for end address of the range, can be NULL

int *out_nid
ptr to int for nid of the range, can be NULL

Description

Finds the next range from type_a which is not marked as unsuitable in type_b.

Reverse of *__next_mem_range()*.

int __init_memblock **memblock_set_node**(phys_addr_t base, phys_addr_t size, struct *memblock_type* *type, int nid)

set node ID on memblock regions

Parameters

phys_addr_t base
base of area to set node ID for

phys_addr_t size
size of area to set node ID for

struct memblock_type *type
memblock type to set node ID for

int nid
node ID to set

Description

Set the nid of memblock **type** regions in [**base**, **base** + **size**) to **nid**. Regions which cross the area boundaries are split as necessary.

Return

0 on success, -errno on failure.

void __init_memblock **__next_mem_pfn_range_in_zone**(u64 *idx, struct *zone* *zone, unsigned long *out_spfn, unsigned long *out_epfn)

iterator for for_each_*_range_in_zone()

Parameters

u64 *idx
pointer to u64 loop variable

struct zone *zone
zone in which all of the memory blocks reside

unsigned long *out_spfn
ptr to ulong for start pfn of the range, can be `NULL`

unsigned long *out_epfn
ptr to ulong for end pfn of the range, can be `NULL`

Description

This function is meant to be a zone/pfn specific wrapper for the for_each_mem_range type iterators. Specifically they are used in the deferred memory init routines and as such we were duplicating much of this logic throughout the code. So instead of having it in multiple locations it seemed like it would make more sense to centralize this to one new iterator that does everything they need.

phys_addr_t **memblock_alloc_range_nid**(phys_addr_t size, phys_addr_t align, phys_addr_t start, phys_addr_t end, int nid, bool exact_nid)

allocate boot memory block

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t start
the lower bound of the memory region to allocate (phys address)

phys_addr_t end
the upper bound of the memory region to allocate (phys address)

int nid
nid of the free area to find, `NUMA_NO_NODE` for any node

bool exact_nid
control the allocation fall back to other nodes

Description

The allocation is performed from memory region limited by memblock.current_limit if **end** == `MEMBLOCK_ALLOC_ACCESSIBLE`.

If the specified node can not hold the requested memory and **exact_nid** is false, the allocation falls back to any node in the system.

For systems with memory mirroring, the allocation is attempted first from the regions with mirroring enabled and then retried from any memory region.

In addition, function using kmemleak_alloc_phys for allocated boot memory block, it is never reported as leaks.

Return

Physical address of allocated memory block on success, `0` on failure.

phys_addr_t **memblock_phys_alloc_range**(phys_addr_t size, phys_addr_t align, phys_addr_t start, phys_addr_t end)

allocate a memory block inside specified range

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t start
the lower bound of the memory region to allocate (physical address)

phys_addr_t end
the upper bound of the memory region to allocate (physical address)

Description

Allocate **size** bytes in the between **start** and **end**.

Return

physical address of the allocated memory block on success, `0` on failure.

phys_addr_t **memblock_phys_alloc_try_nid**(phys_addr_t size, phys_addr_t align, int nid)

allocate a memory block from specified NUMA node

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

int nid
nid of the free area to find, `NUMA_NO_NODE` for any node

Description

Allocates memory block from the specified NUMA node. If the node has no available memory, attempts to allocated from any node in the system.

Return

physical address of the allocated memory block on success, `0` on failure.

void ***memblock_alloc_internal**(phys_addr_t size, phys_addr_t align, phys_addr_t min_addr, phys_addr_t max_addr, int nid, bool exact_nid)

allocate boot memory block

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t min_addr
the lower bound of the memory region to allocate (phys address)

phys_addr_t max_addr
the upper bound of the memory region to allocate (phys address)

int nid
nid of the free area to find, NUMA_NO_NODE for any node

bool exact_nid
control the allocation fall back to other nodes

Description

Allocates memory block using *memblock_alloc_range_nid()* and converts the returned physical address to virtual.

The **min_addr** limit is dropped if it can not be satisfied and the allocation will fall back to memory below **min_addr**. Other constraints, such as node and mirrored memory will be handled again in *memblock_alloc_range_nid()*.

Return

Virtual address of allocated memory block on success, NULL on failure.

void ***memblock_alloc_exact_nid_raw**(phys_addr_t size, phys_addr_t align, phys_addr_t min_addr, phys_addr_t max_addr, int nid)

allocate boot memory block on the exact node without zeroing memory

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t min_addr
the lower bound of the memory region from where the allocation is preferred (phys address)

phys_addr_t max_addr
the upper bound of the memory region from where the allocation is preferred (phys address), or MEMBLOCK_ALLOC_ACCESSIBLE to allocate only from memory limited by memblock.current_limit value

int nid
nid of the free area to find, NUMA_NO_NODE for any node

Description

Public function, provides additional debug information (including caller info), if enabled. Does not zero allocated memory.

Return

Virtual address of allocated memory block on success, NULL on failure.

void ***memblock_alloc_try_nid_raw**(phys_addr_t size, phys_addr_t align, phys_addr_t min_addr, phys_addr_t max_addr, int nid)

allocate boot memory block without zeroing memory and without panicking

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t min_addr
the lower bound of the memory region from where the allocation is preferred (phys address)

phys_addr_t max_addr
the upper bound of the memory region from where the allocation is preferred (phys address), or `MEMBLOCK_ALLOC_ACCESSIBLE` to allocate only from memory limited by memblock.current_limit value

int nid
nid of the free area to find, `NUMA_NO_NODE` for any node

Description

Public function, provides additional debug information (including caller info), if enabled. Does not zero allocated memory, does not panic if request cannot be satisfied.

Return

Virtual address of allocated memory block on success, NULL on failure.

void ***memblock_alloc_try_nid**(phys_addr_t size, phys_addr_t align, phys_addr_t min_addr, phys_addr_t max_addr, int nid)

allocate boot memory block

Parameters

phys_addr_t size
size of memory block to be allocated in bytes

phys_addr_t align
alignment of the region and block's size

phys_addr_t min_addr
the lower bound of the memory region from where the allocation is preferred (phys address)

phys_addr_t max_addr
the upper bound of the memory region from where the allocation is preferred (phys address), or `MEMBLOCK_ALLOC_ACCESSIBLE` to allocate only from memory limited by memblock.current_limit value

int nid
nid of the free area to find, `NUMA_NO_NODE` for any node

Description

Public function, provides additional debug information (including caller info), if enabled. This function zeroes the allocated memory.

Return

Virtual address of allocated memory block on success, NULL on failure.

void **memblock_free_late**(phys_addr_t base, phys_addr_t size)
free pages directly to buddy allocator

Parameters

phys_addr_t base
phys starting address of the boot memory block

phys_addr_t size
size of the boot memory block in bytes

Description

This is only useful when the memblock allocator has already been torn down, but we are still initializing the system. Pages are released directly to the buddy allocator.

bool __init_memblock **memblock_is_region_memory**(phys_addr_t base, phys_addr_t size)
check if a region is a subset of memory

Parameters

phys_addr_t base
base of region to check

phys_addr_t size
size of region to check

Description

Check if the region [**base**, **base** + **size**) is a subset of a memory block.

Return

0 if false, non-zero if true

bool __init_memblock **memblock_is_region_reserved**(phys_addr_t base, phys_addr_t size)
check if a region intersects reserved memory

Parameters

phys_addr_t base
base of region to check

phys_addr_t size
size of region to check

Description

Check if the region [**base**, **base** + **size**) intersects a reserved memory block.

Return

True if they intersect, false if not.

void **memblock_free_all**(void)
release free pages to the buddy allocator

Parameters

void
no arguments

6.11 GFP masks used from FS/IO context

Date
May, 2018

Author
Michal Hocko <mhocko@kernel.org>

6.11.1 Introduction

Code paths in the filesystem and IO stacks must be careful when allocating memory to prevent recursion deadlocks caused by direct memory reclaim calling back into the FS or IO paths and blocking on already held resources (e.g. locks - most commonly those used for the transaction context).

The traditional way to avoid this deadlock problem is to clear __GFP_FS respectively __GFP_IO (note the latter implies clearing the first as well) in the gfp mask when calling an allocator. GFP_NOFS respectively GFP_NOIO can be used as shortcut. It turned out though that above approach has led to abuses when the restricted gfp mask is used “just in case” without a deeper consideration which leads to problems because an excessive use of GFP_NOFS/GFP_NOIO can lead to memory over-reclaim or other memory reclaim issues.

6.11.2 New API

Since 4.12 we do have a generic scope API for both NOFS and NOIO context `memalloc_nofs_save`, `memalloc_nofs_restore` respectively `memalloc_noio_save`, `memalloc_noio_restore` which allow to mark a scope to be a critical section from a filesystem or I/O point of view. Any allocation from that scope will inherently drop __GFP_FS respectively __GFP_IO from the given mask so no memory allocation can recurse back in the FS/IO.

unsigned int **memalloc_nofs_save**(void)
Marks implicit GFP_NOFS allocation scope.

Parameters

void
no arguments

Description

This functions marks the beginning of the GFP_NOFS allocation scope. All further allocations will implicitly drop __GFP_FS flag and so they are safe for the FS critical section from the allocation recursion point of view. Use memalloc_nofs_restore to end the scope with flags returned by this function.

This function is safe to be used from any context.

void **memalloc_nofs_restore**(unsigned int flags)
Ends the implicit GFP_NOFS scope.

Parameters

unsigned int flags
Flags to restore.

Description

Ends the implicit GFP_NOFS scope started by memalloc_nofs_save function. Always make sure that the given flags is the return value from the pairing memalloc_nofs_save call.

unsigned int **memalloc_noio_save**(void)

Marks implicit GFP_NOIO allocation scope.

Parameters

void
no arguments

Description

This functions marks the beginning of the GFP_NOIO allocation scope. All further allocations will implicitly drop __GFP_IO flag and so they are safe for the IO critical section from the allocation recursion point of view. Use memalloc_noio_restore to end the scope with flags returned by this function.

This function is safe to be used from any context.

void **memalloc_noio_restore**(unsigned int flags)

Ends the implicit GFP_NOIO scope.

Parameters

unsigned int flags
Flags to restore.

Description

Ends the implicit GFP_NOIO scope started by memalloc_noio_save function. Always make sure that the given flags is the return value from the pairing memalloc_noio_save call.

FS/IO code then simply calls the appropriate save function before any critical section with respect to the reclaim is started - e.g. lock shared with the reclaim context or when a transaction context nesting would be possible via reclaim. The restore function should be called when the critical section ends. All that ideally along with an explanation what is the reclaim context for easier maintenance.

Please note that the proper pairing of save/restore functions allows nesting so it is safe to call `memalloc_noio_save` or `memalloc_noio_restore` respectively from an existing NOIO or NOFS scope.

6.11.3 What about __vmalloc(GFP_NOFS)

vmalloc doesn't support GFP_NOFS semantic because there are hardcoded GFP_KERNEL allocations deep inside the allocator which are quite non-trivial to fix up. That means that calling `vmalloc` with GFP_NOFS/GFP_NOIO is almost always a bug. The good news is that the NOFS/NOIO semantic can be achieved by the scope API.

In the ideal world, upper layers should already mark dangerous contexts and so no special care is required and vmalloc should be called without any problems. Sometimes if the context is not really clear or there are layering violations then the recommended way around that is to wrap `vmalloc` by the scope API with a comment explaining the problem.

CHAPTER

SEVEN

INTERFACES FOR KERNEL DEBUGGING

7.1 The object-lifetime debugging infrastructure

Author
Thomas Gleixner

7.1.1 Introduction

debugobjects is a generic infrastructure to track the life time of kernel objects and validate the operations on those.

debugobjects is useful to check for the following error patterns:

- Activation of uninitialized objects
- Initialization of active objects
- Usage of freed/destroyed objects

debugobjects is not changing the data structure of the real object so it can be compiled in with a minimal runtime impact and enabled on demand with a kernel command line option.

7.1.2 Howto use debugobjects

A kernel subsystem needs to provide a data structure which describes the object type and add calls into the debug code at appropriate places. The data structure to describe the object type needs at minimum the name of the object type. Optional functions can and should be provided to fixup detected problems so the kernel can continue to work and the debug information can be retrieved from a live system instead of hard core debugging with serial consoles and stack trace transcripts from the monitor.

The debug calls provided by debugobjects are:

- debug_object_init
- debug_object_init_on_stack
- debug_object_activate
- debug_object_deactivate
- debug_object_destroy
- debug_object_free

- debug_object_assert_init

Each of these functions takes the address of the real object and a pointer to the object type specific debug description structure.

Each detected error is reported in the statistics and a limited number of errors are printk'ed including a full stack trace.

The statistics are available via /sys/kernel/debug/debug_objects/stats. They provide information about the number of warnings and the number of successful fixups along with information about the usage of the internal tracking objects and the state of the internal tracking objects pool.

7.1.3 Debug functions

void **debug_object_init**(void *addr, const struct *debug_obj_descr* *descr)

debug checks when an object is initialized

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called whenever the initialization function of a real object is called.

When the real object is already tracked by debugobjects it is checked, whether the object can be initialized. Initializing is not allowed for active and destroyed objects. When debugobjects detects an error, then it calls the fixup_init function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real initialization of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

When the real object is not yet tracked by debugobjects, debugobjects allocates a tracker object for the real object and sets the tracker object state to ODEBUG_STATE_INIT. It verifies that the object is not on the callers stack. If it is on the callers stack then a limited number of warnings including a full stack trace is printk'ed. The calling code must use `debug_object_init_on_stack()` and remove the object before leaving the function which allocated it. See next section.

void **debug_object_init_on_stack**(void *addr, const struct *debug_obj_descr* *descr)

debug checks when an object on stack is initialized

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called whenever the initialization function of a real object which resides on the stack is called.

When the real object is already tracked by debugobjects it is checked, whether the object can be initialized. Initializing is not allowed for active and destroyed objects. When debugobjects detects an error, then it calls the fixup_init function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real initialization of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

When the real object is not yet tracked by debugobjects debugobjects allocates a tracker object for the real object and sets the tracker object state to ODEBUG_STATE_INIT. It verifies that the object is on the callers stack.

An object which is on the stack must be removed from the tracker by calling *debug_object_free()* before the function which allocates the object returns. Otherwise we keep track of stale objects.

int **debug_object_activate**(void *addr, const struct *debug_obj_descr* *descr)

debug checks when an object is activated

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure Returns 0 for success, -EINVAL for check failed.

This function is called whenever the activation function of a real object is called.

When the real object is already tracked by debugobjects it is checked, whether the object can be activated. Activating is not allowed for active and destroyed objects. When debugobjects detects an error, then it calls the fixup_activate function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real activation of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

When the real object is not yet tracked by debugobjects then the fixup_activate function is called if available. This is necessary to allow the legitimate activation of statically allocated and initialized objects. The fixup function checks whether the object is valid and calls the debug_objects_init() function to initialize the tracking of this object.

When the activation is legitimate, then the state of the associated tracker object is set to ODEBUG_STATE_ACTIVE.

void **debug_object_deactivate**(void *addr, const struct *debug_obj_descr* *descr)

debug checks when an object is deactivated

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called whenever the deactivation function of a real object is called.

When the real object is tracked by debugobjects it is checked, whether the object can be deactivated. Deactivating is not allowed for untracked or destroyed objects.

When the deactivation is legitimate, then the state of the associated tracker object is set to ODEBUG_STATE_INACTIVE.

void **debug_object_destroy**(void *addr, const struct *debug_obj_descr* *descr)
debug checks when an object is destroyed

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called to mark an object destroyed. This is useful to prevent the usage of invalid objects, which are still available in memory: either statically allocated objects or objects which are freed later.

When the real object is tracked by debugobjects it is checked, whether the object can be destroyed. Destruction is not allowed for active and destroyed objects. When debugobjects detects an error, then it calls the fixup_destroy function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real destruction of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

When the destruction is legitimate, then the state of the associated tracker object is set to ODEBUG_STATE_DESTROYED.

void **debug_object_free**(void *addr, const struct *debug_obj_descr* *descr)
debug checks when an object is freed

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called before an object is freed.

When the real object is tracked by debugobjects it is checked, whether the object can be freed. Free is not allowed for active objects. When debugobjects detects an error, then it calls the fixup_free function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real free of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

Note that debug_object_free removes the object from the tracker. Later usage of the object is detected by the other debug checks.

void **debug_object_assert_init**(void *addr, const struct *debug_obj_descr* *descr)
debug checks when object should be init-ed

Parameters

void *addr
address of the object

const struct debug_obj_descr *descr
pointer to an object specific debug description structure

This function is called to assert that an object has been initialized.

When the real object is not tracked by debugobjects, it calls fixup_assert_init of the object type description structure provided by the caller, with the hardcoded object state ODEBUG_NOT_AVAILABLE. The fixup function can correct the problem by calling debug_object_init and other specific initializing functions.

When the real object is already tracked by debugobjects it is ignored.

7.1.4 Fixup functions

Debug object type description structure

struct **debug_obj**
  representation of an tracked object

Definition:

```
struct debug_obj {
    struct hlist_node       node;
    enum debug_obj_state    state;
    unsigned int            astate;
    void *object;
    const struct debug_obj_descr *descr;
};
```

Members

node
  hlist node to link the object into the tracker list

state
  tracked object state

astate
  current active state

object
  pointer to the real object

descr
  pointer to an object type specific debug description structure

struct **debug_obj_descr**
  object type specific debug description structure

Definition:

```
struct debug_obj_descr {
    const char              *name;
    void *(*debug_hint)(void *addr);
    bool (*is_static_object)(void *addr);
    bool (*fixup_init)(void *addr, enum debug_obj_state state);
    bool (*fixup_activate)(void *addr, enum debug_obj_state state);
    bool (*fixup_destroy)(void *addr, enum debug_obj_state state);
    bool (*fixup_free)(void *addr, enum debug_obj_state state);
```

```
    bool (*fixup_assert_init)(void *addr, enum debug_obj_state state);
};
```

Members

name
name of the object typee

debug_hint
function returning address, which have associated kernel symbol, to allow identify the object

is_static_object
return true if the obj is static, otherwise return false

fixup_init
fixup function, which is called when the init check fails. All fixup functions must return true if fixup was successful, otherwise return false

fixup_activate
fixup function, which is called when the activate check fails

fixup_destroy
fixup function, which is called when the destroy check fails

fixup_free
fixup function, which is called when the free check fails

fixup_assert_init
fixup function, which is called when the assert_init check fails

fixup_init

This function is called from the debug code whenever a problem in debug_object_init is detected. The function takes the address of the object and the state which is currently recorded in the tracker.

Called from debug_object_init when the object state is:

- ODEBUG_STATE_ACTIVE

The function returns true when the fixup was successful, otherwise false. The return value is used to update the statistics.

Note, that the function needs to call the *debug_object_init()* function again, after the damage has been repaired in order to keep the state consistent.

fixup_activate

This function is called from the debug code whenever a problem in debug_object_activate is detected.

Called from debug_object_activate when the object state is:

- ODEBUG_STATE_NOTAVAILABLE
- ODEBUG_STATE_ACTIVE

The function returns true when the fixup was successful, otherwise false. The return value is used to update the statistics.

Note that the function needs to call the *debug_object_activate()* function again after the damage has been repaired in order to keep the state consistent.

The activation of statically initialized objects is a special case. When *debug_object_activate()* has no tracked object for this object address then fixup_activate() is called with object state ODEBUG_STATE_NOTAVAILABLE. The fixup function needs to check whether this is a legitimate case of a statically initialized object or not. In case it is it calls *debug_object_init()* and *debug_object_activate()* to make the object known to the tracker and marked active. In this case the function should return false because this is not a real fixup.

fixup_destroy

This function is called from the debug code whenever a problem in debug_object_destroy is detected.

Called from debug_object_destroy when the object state is:

- ODEBUG_STATE_ACTIVE

The function returns true when the fixup was successful, otherwise false. The return value is used to update the statistics.

fixup_free

This function is called from the debug code whenever a problem in debug_object_free is detected. Further it can be called from the debug checks in kfree/vfree, when an active object is detected from the debug_check_no_obj_freed() sanity checks.

Called from *debug_object_free()* or debug_check_no_obj_freed() when the object state is:

- ODEBUG_STATE_ACTIVE

The function returns true when the fixup was successful, otherwise false. The return value is used to update the statistics.

fixup_assert_init

This function is called from the debug code whenever a problem in debug_object_assert_init is detected.

Called from *debug_object_assert_init()* with a hardcoded state ODEBUG_STATE_NOTAVAILABLE when the object is not found in the debug bucket.

The function returns true when the fixup was successful, otherwise false. The return value is used to update the statistics.

Note, this function should make sure *debug_object_init()* is called before returning.

The handling of statically initialized objects is a special case. The fixup function should check if this is a legitimate case of a statically initialized object or not. In this case only *debug_object_init()* should be called to make the object known to the tracker. Then the function should return false because this is not a real fixup.

7.1.5 Known Bugs And Assumptions

None (knock on wood).

7.2 The Linux Kernel Tracepoint API

Author
Jason Baron

Author
William Cohen

7.2.1 Introduction

Tracepoints are static probe points that are located in strategic points throughout the kernel. 'Probes' register/unregister with tracepoints via a callback mechanism. The 'probes' are strictly typed functions that are passed a unique set of parameters defined by each tracepoint.

From this simple callback mechanism, 'probes' can be used to profile, debug, and understand kernel behavior. There are a number of tools that provide a framework for using 'probes'. These tools include Systemtap, ftrace, and LTTng.

Tracepoints are defined in a number of header files via various macros. Thus, the purpose of this document is to provide a clear accounting of the available tracepoints. The intention is to understand not only what tracepoints are available but also to understand where future tracepoints might be added.

The API presented has functions of the form: `trace_tracepointname(function parameters)`. These are the tracepoints callbacks that are found throughout the code. Registering and unregistering probes with these callback sites is covered in the `Documentation/trace/*` directory.

7.2.2 IRQ

void **trace_irq_handler_entry**(int irq, struct *irqaction* *action)

called immediately before the irq action handler

Parameters

int irq
irq number

struct irqaction *action
pointer to *struct irqaction*

Description

The *struct irqaction* pointed to by **action** contains various information about the handler, including the device name, **action->name**, and the device id, **action->dev_id**. When used in conjunction with the irq_handler_exit tracepoint, we can figure out irq handler latencies.

void **trace_irq_handler_exit**(int irq, struct *irqaction* *action, int ret)

called immediately after the irq action handler returns

Parameters

int irq
irq number

struct irqaction *action
pointer to *struct irqaction*

int ret
return value

Description

If the **ret** value is set to IRQ_HANDLED, then we know that the corresponding **action->handler** successfully handled this irq. Otherwise, the irq might be a shared irq line, or the irq was not handled successfully. Can be used in conjunction with the irq_handler_entry to understand irq handler latencies.

void **trace_softirq_entry**(unsigned int vec_nr)

called immediately before the softirq handler

Parameters

unsigned int vec_nr
softirq vector number

Description

When used in combination with the softirq_exit tracepoint we can determine the softirq handler routine.

void **trace_softirq_exit**(unsigned int vec_nr)

called immediately after the softirq handler returns

Parameters

unsigned int vec_nr
softirq vector number

Description

When used in combination with the softirq_entry tracepoint we can determine the softirq handler routine.

void **trace_softirq_raise**(unsigned int vec_nr)

called immediately when a softirq is raised

Parameters

unsigned int vec_nr
softirq vector number

Description

When used in combination with the softirq_entry tracepoint we can determine the softirq raise to run latency.

void **trace_tasklet_entry**(struct tasklet_struct *t, void *func)

called immediately before the tasklet is run

Parameters

struct tasklet_struct *t
tasklet pointer

void *func
tasklet callback or function being run

Description

Used to find individual tasklet execution time

void **trace_tasklet_exit**(struct tasklet_struct *t, void *func)

called immediately after the tasklet is run

Parameters

struct tasklet_struct *t
tasklet pointer

void *func
tasklet callback or function being run

Description

Used to find individual tasklet execution time

7.2.3 SIGNAL

void **trace_signal_generate**(int sig, struct kernel_siginfo *info, struct task_struct *task, int group, int result)

called when a signal is generated

Parameters

int sig
signal number

struct kernel_siginfo *info
pointer to struct siginfo

struct task_struct *task
pointer to struct task_struct

int group
shared or private

int result
TRACE_SIGNAL_*

Description

Current process sends a 'sig' signal to 'task' process with 'info' siginfo. If 'info' is SEND_SIG_NOINFO or SEND_SIG_PRIV, 'info' is not a pointer and you can't access its field. Instead, SEND_SIG_NOINFO means that si_code is SI_USER, and SEND_SIG_PRIV means that si_code is SI_KERNEL.

void **trace_signal_deliver**(int sig, struct kernel_siginfo *info, struct k_sigaction *ka)
called when a signal is delivered

Parameters

int sig
signal number

struct kernel_siginfo *info
pointer to struct siginfo

struct k_sigaction *ka
pointer to struct k_sigaction

Description

A 'sig' signal is delivered to current process with 'info' siginfo, and it will be handled by 'ka'. ka->sa.sa_handler can be SIG_IGN or SIG_DFL. Note that some signals reported by signal_generate tracepoint can be lost, ignored or modified (by debugger) before hitting this tracepoint. This means, this can show which signals are actually delivered, but matching generated signals and delivered signals may not be correct.

7.2.4 Block IO

void **trace_block_touch_buffer**(struct buffer_head *bh)
mark a buffer accessed

Parameters

struct buffer_head *bh
buffer_head being touched

Description

Called from touch_buffer().

void **trace_block_dirty_buffer**(struct buffer_head *bh)
mark a buffer dirty

Parameters

struct buffer_head *bh
    buffer_head being dirtied

Description

Called from mark_buffer_dirty().

void **trace_block_rq_requeue**(struct request *rq)
    place block IO request back on a queue

Parameters

struct request *rq
    block IO operation request

Description

The block operation request **rq** is being placed back into queue **q**. For some reason the request was not completed and needs to be put back in the queue.

void **trace_block_rq_complete**(struct request *rq, blk_status_t error, unsigned int nr_bytes)
    block IO operation completed by device driver

Parameters

struct request *rq
    block operations request

blk_status_t error
    status code

unsigned int nr_bytes
    number of completed bytes

Description

The block_rq_complete tracepoint event indicates that some portion of operation request has been completed by the device driver. If the **rq->bio** is `NULL`, then there is absolutely no additional work to do for the request. If **rq->bio** is non-NULL then there is additional work required to complete the request.

void **trace_block_rq_error**(struct request *rq, blk_status_t error, unsigned int nr_bytes)
    block IO operation error reported by device driver

Parameters

struct request *rq
    block operations request

blk_status_t error
    status code

unsigned int nr_bytes
    number of completed bytes

Description

The block_rq_error tracepoint event indicates that some portion of operation request has failed as reported by the device driver.

void **trace_block_rq_insert**(struct request *rq)
insert block operation request into queue

Parameters

struct request *rq
block IO operation request

Description

Called immediately before block operation request **rq** is inserted into queue **q**. The fields in the operation request **rq** struct can be examined to determine which device and sectors the pending operation would access.

void **trace_block_rq_issue**(struct request *rq)
issue pending block IO request operation to device driver

Parameters

struct request *rq
block IO operation request

Description

Called when block operation request **rq** from queue **q** is sent to a device driver for processing.

void **trace_block_rq_merge**(struct request *rq)
merge request with another one in the elevator

Parameters

struct request *rq
block IO operation request

Description

Called when block operation request **rq** from queue **q** is merged to another request queued in the elevator.

void **trace_block_io_start**(struct request *rq)
insert a request for execution

Parameters

struct request *rq
block IO operation request

Description

Called when block operation request **rq** is queued for execution

void **trace_block_io_done**(struct request *rq)
block IO operation request completed

Parameters

struct request *rq
block IO operation request

Description

Called when block operation request **rq** is completed

void **trace_block_bio_complete**(struct request_queue *q, struct *bio* *bio)
completed all work on the block operation

Parameters

struct request_queue *q
queue holding the block operation

struct bio *bio
block operation completed

Description

This tracepoint indicates there is no further work to do on this block IO operation **bio**.

void **trace_block_bio_bounce**(struct *bio* *bio)
used bounce buffer when processing block operation

Parameters

struct bio *bio
block operation

Description

A bounce buffer was used to handle the block operation **bio** in **q**. This occurs when hardware limitations prevent a direct transfer of data between the **bio** data memory area and the IO device. Use of a bounce buffer requires extra copying of data and decreases performance.

void **trace_block_bio_backmerge**(struct *bio* *bio)
merging block operation to the end of an existing operation

Parameters

struct bio *bio
new block operation to merge

Description

Merging block request **bio** to the end of an existing block request.

void **trace_block_bio_frontmerge**(struct *bio* *bio)
merging block operation to the beginning of an existing operation

Parameters

struct bio *bio
new block operation to merge

Description

Merging block IO operation **bio** to the beginning of an existing block request.

void **trace_block_bio_queue**(struct *bio* *bio)
putting new block IO operation in queue

Parameters

struct bio *bio
new block operation

Description

About to place the block IO operation **bio** into queue **q**.

void **trace_block_getrq**(struct *bio* *bio)

get a free request entry in queue for block IO operations

Parameters

struct bio *bio
pending block IO operation (can be NULL)

Description

A request struct has been allocated to handle the block IO operation **bio**.

void **trace_block_plug**(struct request_queue *q)

keep operations requests in request queue

Parameters

struct request_queue *q
request queue to plug

Description

Plug the request queue **q**. Do not allow block operation requests to be sent to the device driver. Instead, accumulate requests in the queue to improve throughput performance of the block device.

void **trace_block_unplug**(struct request_queue *q, unsigned int depth, bool explicit)

release of operations requests in request queue

Parameters

struct request_queue *q
request queue to unplug

unsigned int depth
number of requests just added to the queue

bool explicit
whether this was an explicit unplug, or one from schedule()

Description

Unplug request queue **q** because device driver is scheduled to work on elements in the request queue.

void **trace_block_split**(struct *bio* *bio, unsigned int new_sector)

split a single bio struct into two bio structs

Parameters

struct bio *bio
block operation being split

unsigned int new_sector
The starting sector for the new bio

Description

The bio request **bio** needs to be split into two bio requests. The newly created **bio** request starts at **new_sector**. This split may be required due to hardware limitations such as operation crossing device boundaries in a RAID system.

void **trace_block_bio_remap**(struct *bio* *bio, dev_t dev, sector_t from)

map request for a logical device to the raw device

Parameters

struct bio *bio
revised operation

dev_t dev
original device for the operation

sector_t from
original sector for the operation

Description

An operation for a logical device has been mapped to the raw block device.

void **trace_block_rq_remap**(struct request *rq, dev_t dev, sector_t from)

map request for a block operation request

Parameters

struct request *rq
block IO operation request

dev_t dev
device for the operation

sector_t from
original sector for the operation

Description

The block operation request **rq** in **q** has been remapped. The block operation request **rq** holds the current information and **from** hold the original sector.

7.2.5 Workqueue

void **trace_workqueue_queue_work**(int req_cpu, struct pool_workqueue *pwq, struct work_struct *work)

called when a work gets queued

Parameters

int req_cpu
the requested cpu

struct pool_workqueue *pwq
pointer to struct pool_workqueue

struct work_struct *work
pointer to struct work_struct

Description

This event occurs when a work is queued immediately or once a delayed work is actually queued on a workqueue (ie: once the delay has been reached).

void **trace_workqueue_activate_work**(struct work_struct *work)
called when a work gets activated

Parameters

struct work_struct *work
pointer to struct work_struct

Description

This event occurs when a queued work is put on the active queue, which happens immediately after queueing unless **max_active** limit is reached.

void **trace_workqueue_execute_start**(struct work_struct *work)
called immediately before the workqueue callback

Parameters

struct work_struct *work
pointer to struct work_struct

Description

Allows to track workqueue execution.

void **trace_workqueue_execute_end**(struct work_struct *work, work_func_t function)
called immediately after the workqueue callback

Parameters

struct work_struct *work
pointer to struct work_struct

work_func_t function
pointer to worker function

Description

Allows to track workqueue execution.

7.3 Using physical DMA provided by OHCI-1394 FireWire controllers for debugging

7.3.1 Introduction

Basically all FireWire controllers which are in use today are compliant to the OHCI-1394 specification which defines the controller to be a PCI bus master which uses DMA to offload data transfers from the CPU and has a "Physical Response Unit" which executes specific requests by employing PCI-Bus master DMA after applying filters defined by the OHCI-1394 driver.

Once properly configured, remote machines can send these requests to ask the OHCI-1394 controller to perform read and write requests on physical system memory and, for read requests, send the result of the physical memory read back to the requester.

With that, it is possible to debug issues by reading interesting memory locations such as buffers like the printk buffer or the process table.

Retrieving a full system memory dump is also possible over the FireWire, using data transfer rates in the order of 10MB/s or more.

With most FireWire controllers, memory access is limited to the low 4 GB of physical address space. This can be a problem on IA64 machines where memory is located mostly above that limit, but it is rarely a problem on more common hardware such as x86, x86-64 and PowerPC.

At least LSI FW643e and FW643e2 controllers are known to support access to physical addresses above 4 GB, but this feature is currently not enabled by Linux.

Together with a early initialization of the OHCI-1394 controller for debugging, this facility proved most useful for examining long debugs logs in the printk buffer on to debug early boot problems in areas like ACPI where the system fails to boot and other means for debugging (serial port) are either not available (notebooks) or too slow for extensive debug information (like ACPI).

7.3.2 Drivers

The firewire-ohci driver in drivers/firewire uses filtered physical DMA by default, which is more secure but not suitable for remote debugging. Pass the remote_dma=1 parameter to the driver to get unfiltered physical DMA.

Because the firewire-ohci driver depends on the PCI enumeration to be completed, an initialization routine which runs pretty early has been implemented for x86. This routine runs long before console_init() can be called, i.e. before the printk buffer appears on the console.

To activate it, enable CONFIG_PROVIDE_OHCI1394_DMA_INIT (Kernel hacking menu: Remote debugging over FireWire early on boot) and pass the parameter "ohci1394_dma=early" to the recompiled kernel on boot.

7.3.3 Tools

firescope - Originally developed by Benjamin Herrenschmidt, Andi Kleen ported it from PowerPC to x86 and x86_64 and added functionality, firescope can now be used to view the printk buffer of a remote machine, even with live update.

Bernhard Kaindl enhanced firescope to support accessing 64-bit machines from 32-bit firescope and vice versa: - http://v3.sk/~lkundrak/firescope/

and he implemented fast system dump (alpha version - read README.txt): - http://halobates.de/firewire/firedump-0.1.tar.bz2

There is also a gdb proxy for firewire which allows to use gdb to access data which can be referenced from symbols found by gdb in vmlinux: - http://halobates.de/firewire/fireproxy-0.33.tar.bz2

The latest version of this gdb proxy (fireproxy-0.34) can communicate (not yet stable) with kgdb over an memory-based communication module (kgdbom).

7.3.4 Getting Started

The OHCI-1394 specification regulates that the OHCI-1394 controller must disable all physical DMA on each bus reset.

This means that if you want to debug an issue in a system state where interrupts are disabled and where no polling of the OHCI-1394 controller for bus resets takes place, you have to establish any FireWire cable connections and fully initialize all FireWire hardware __before__ the system enters such state.

Step-by-step instructions for using firescope with early OHCI initialization:

1) Verify that your hardware is supported:

   Load the firewire-ohci module and check your kernel logs. You should see a line similar to:

   ```
   firewire_ohci 0000:15:00.1: added OHCI v1.0 device as card 2, 4 IR + 4 IT
   ... contexts, quirks 0x11
   ```

   when loading the driver. If you have no supported controller, many PCI, CardBus and even some Express cards which are fully compliant to OHCI-1394 specification are available. If it requires no driver for Windows operating systems, it most likely is. Only specialized shops have cards which are not compliant, they are based on TI PCILynx chips and require drivers for Windows operating systems.

   The mentioned kernel log message contains the string "physUB" if the controller implements a writable Physical Upper Bound register. This is required for physical DMA above 4 GB (but not utilized by Linux yet).

2) Establish a working FireWire cable connection:

   Any FireWire cable, as long at it provides electrically and mechanically stable connection and has matching connectors (there are small 4-pin and large 6-pin FireWire ports) will do.

   If an driver is running on both machines you should see a line like:

   ```
   firewire_core 0000:15:00.1: created device fw1: GUID 00061b0020105917, S400
   ```

   on both machines in the kernel log when the cable is plugged in and connects the two machines.

3) Test physical DMA using firescope:

   On the debug host, make sure that /dev/fw* is accessible, then start firescope:

   ```
       $ firescope
       Port 0 (/dev/fw1) opened, 2 nodes detected

       FireScope
       ---------
       Target : <unspecified>
       Gen    : 1
       [Ctrl-T] choose target
       [Ctrl-H] this menu
       [Ctrl-Q] quit
   ```

```
------> Press Ctrl-T now, the output should be similar to:

    2 nodes available, local node is: 0
     0: ffc0, uuid: 00000000 00000000 [LOCAL]
     1: ffc1, uuid: 00279000 ba4bb801
```

Besides the [LOCAL] node, it must show another node without error message.

4) Prepare for debugging with early OHCI-1394 initialization:

   4.1) Kernel compilation and installation on debug target

   Compile the kernel to be debugged with CONFIG_PROVIDE_OHCI1394_DMA_INIT (Kernel hacking: Provide code for enabling DMA over FireWire early on boot) enabled and install it on the machine to be debugged (debug target).

   4.2) Transfer the System.map of the debugged kernel to the debug host

   Copy the System.map of the kernel be debugged to the debug host (the host which is connected to the debugged machine over the FireWire cable).

5) Retrieving the printk buffer contents:

   With the FireWire cable connected, the OHCI-1394 driver on the debugging host loaded, reboot the debugged machine, booting the kernel which has CONFIG_PROVIDE_OHCI1394_DMA_INIT enabled, with the option ohci1394_dma=early.

   Then, on the debugging host, run firescope, for example by using -A:

```
firescope -A System.map-of-debug-target-kernel
```

   Note: -A automatically attaches to the first non-local node. It only works reliably if only connected two machines are connected using FireWire.

   After having attached to the debug target, press Ctrl-D to view the complete printk buffer or Ctrl-U to enter auto update mode and get an updated live view of recent kernel messages logged on the debug target.

   Call “firescope -h” to get more information on firescope’s options.

7.3.5 Notes

Documentation and specifications: http://halobates.de/firewire/

FireWire is a trademark of Apple Inc. - for more information please refer to: https://en.wikipedia.org/wiki/FireWire

CHAPTER

EIGHT

EVERYTHING ELSE

Documents that don't fit elsewhere or which have yet to be categorized.

8.1 Reed-Solomon Library Programming Interface

Author
Thomas Gleixner

8.1.1 Introduction

The generic Reed-Solomon Library provides encoding, decoding and error correction functions.

Reed-Solomon codes are used in communication and storage applications to ensure data integrity.

This documentation is provided for developers who want to utilize the functions provided by the library.

8.1.2 Known Bugs And Assumptions

None.

8.1.3 Usage

This chapter provides examples of how to use the library.

Initializing

The init function init_rs returns a pointer to an rs decoder structure, which holds the necessary information for encoding, decoding and error correction with the given polynomial. It either uses an existing matching decoder or creates a new one. On creation all the lookup tables for fast en/decoding are created. The function may take a while, so make sure not to call it in critical code paths.

```
/* the Reed Solomon control structure */
static struct rs_control *rs_decoder;
```

```
/* Symbolsize is 10 (bits)
 * Primitive polynomial is x^10+x^3+1
 * first consecutive root is 0
 * primitive element to generate roots = 1
 * generator polynomial degree (number of roots) = 6
 */
rs_decoder = init_rs (10, 0x409, 0, 1, 6);
```

Encoding

The encoder calculates the Reed-Solomon code over the given data length and stores the result in the parity buffer. Note that the parity buffer must be initialized before calling the encoder.

The expanded data can be inverted on the fly by providing a non-zero inversion mask. The expanded data is XOR'ed with the mask. This is used e.g. for FLASH ECC, where the all 0xFF is inverted to an all 0x00. The Reed-Solomon code for all 0x00 is all 0x00. The code is inverted before storing to FLASH so it is 0xFF too. This prevents that reading from an erased FLASH results in ECC errors.

The databytes are expanded to the given symbol size on the fly. There is no support for encoding continuous bitstreams with a symbol size != 8 at the moment. If it is necessary it should be not a big deal to implement such functionality.

```
/* Parity buffer. Size = number of roots */
uint16_t par[6];
/* Initialize the parity buffer */
memset(par, 0, sizeof(par));
/* Encode 512 byte in data8. Store parity in buffer par */
encode_rs8 (rs_decoder, data8, 512, par, 0);
```

Decoding

The decoder calculates the syndrome over the given data length and the received parity symbols and corrects errors in the data.

If a syndrome is available from a hardware decoder then the syndrome calculation is skipped.

The correction of the data buffer can be suppressed by providing a correction pattern buffer and an error location buffer to the decoder. The decoder stores the calculated error location and the correction bitmask in the given buffers. This is useful for hardware decoders which use a weird bit ordering scheme.

The databytes are expanded to the given symbol size on the fly. There is no support for decoding continuous bitstreams with a symbolsize != 8 at the moment. If it is necessary it should be not a big deal to implement such functionality.

Decoding with syndrome calculation, direct data correction

```
/* Parity buffer. Size = number of roots */
uint16_t par[6];
uint8_t  data[512];
int numerr;
/* Receive data */
.....
/* Receive parity */
.....
/* Decode 512 byte in data8.*/
numerr = decode_rs8 (rs_decoder, data8, par, 512, NULL, 0, NULL, 0, NULL);
```

Decoding with syndrome given by hardware decoder, direct data correction

```
/* Parity buffer. Size = number of roots */
uint16_t par[6], syn[6];
uint8_t  data[512];
int numerr;
/* Receive data */
.....
/* Receive parity */
.....
/* Get syndrome from hardware decoder */
.....
/* Decode 512 byte in data8.*/
numerr = decode_rs8 (rs_decoder, data8, par, 512, syn, 0, NULL, 0, NULL);
```

Decoding with syndrome given by hardware decoder, no direct data correction.

Note: It's not necessary to give data and received parity to the decoder.

```
/* Parity buffer. Size = number of roots */
uint16_t par[6], syn[6], corr[8];
uint8_t  data[512];
int numerr, errpos[8];
/* Receive data */
.....
/* Receive parity */
.....
/* Get syndrome from hardware decoder */
.....
/* Decode 512 byte in data8.*/
numerr = decode_rs8 (rs_decoder, NULL, NULL, 512, syn, 0, errpos, 0, corr);
for (i = 0; i < numerr; i++) {
    do_error_correction_in_your_buffer(errpos[i], corr[i]);
}
```

Cleanup

The function free_rs frees the allocated resources, if the caller is the last user of the decoder.

```
/* Release resources */
free_rs(rs_decoder);
```

8.1.4 Structures

This chapter contains the autogenerated documentation of the structures which are used in the Reed-Solomon Library and are relevant for a developer.

struct **rs_codec**

rs codec data

Definition:

```
struct rs_codec {
    int mm;
    int nn;
    uint16_t *alpha_to;
    uint16_t *index_of;
    uint16_t *genpoly;
    int nroots;
    int fcr;
    int prim;
    int iprim;
    int gfpoly;
    int (*gffunc)(int);
    int users;
    struct list_head list;
};
```

Members

mm
Bits per symbol

nn
Symbols per block (= (1<<mm)-1)

alpha_to
log lookup table

index_of
Antilog lookup table

genpoly
Generator polynomial

nroots
Number of generator roots = number of parity symbols

fcr
First consecutive root, index form

prim
Primitive element, index form

iprim
prim-th root of 1, index form

gfpoly
The primitive generator polynominal

gffunc
Function to generate the field, if non-canonical representation

users
Users of this structure

list
List entry for the rs codec list

struct **rs_control**
rs control structure per instance

Definition:

```
struct rs_control {
    struct rs_codec *codec;
    uint16_t buffers[];
};
```

Members

codec
The codec used for this instance

buffers
Internal scratch buffers used in calls to decode_rs()

struct *rs_control* ***init_rs**(int symsize, int gfpoly, int fcr, int prim, int nroots)
Create a RS control struct and initialize it

Parameters

int symsize
the symbol size (number of bits)

int gfpoly
the extended Galois field generator polynomial coefficients, with the 0th coefficient in the low order bit. The polynomial must be primitive;

int fcr
the first consecutive root of the rs code generator polynomial in index form

int prim
primitive element to generate polynomial roots

int nroots
RS code generator polynomial degree (number of roots)

Description

Allocations use GFP_KERNEL.

8.1.5 Public Functions Provided

This chapter contains the autogenerated documentation of the Reed-Solomon functions which are exported.

void **free_rs**(struct *rs_control* *rs)
Free the rs control structure

Parameters

struct rs_control *rs
The control structure which is not longer used by the caller

Description

Free the control structure. If **rs** is the last user of the associated codec, free the codec as well.

struct *rs_control* ***init_rs_gfp**(int symsize, int gfpoly, int fcr, int prim, int nroots, gfp_t gfp)
Create a RS control struct and initialize it

Parameters

int symsize
the symbol size (number of bits)

int gfpoly
the extended Galois field generator polynomial coefficients, with the 0th coefficient in the low order bit. The polynomial must be primitive;

int fcr
the first consecutive root of the rs code generator polynomial in index form

int prim
primitive element to generate polynomial roots

int nroots
RS code generator polynomial degree (number of roots)

gfp_t gfp
Memory allocation flags.

struct *rs_control* ***init_rs_non_canonical**(int symsize, int (*gffunc)(int), int fcr, int prim, int nroots)
Allocate rs control struct for fields with non-canonical representation

Parameters

int symsize
the symbol size (number of bits)

int (*gffunc)(int)
pointer to function to generate the next field element, or the multiplicative identity element if given 0. Used instead of gfpoly if gfpoly is 0

int fcr
the first consecutive root of the rs code generator polynomial in index form

int prim
primitive element to generate polynomial roots

int nroots
RS code generator polynomial degree (number of roots)

int **encode_rs8**(struct *rs_control* *rsc, uint8_t *data, int len, uint16_t *par, uint16_t invmsk)
Calculate the parity for data values (8bit data width)

Parameters

struct rs_control *rsc
the rs control structure

uint8_t *data
data field of a given type

int len
data length

uint16_t *par
parity data, must be initialized by caller (usually all 0)

uint16_t invmsk
invert data mask (will be xored on data)

The parity uses a uint16_t data type to enable symbol size > 8. The calling code must take care of encoding of the syndrome result for storage itself.

int **decode_rs8**(struct *rs_control* *rsc, uint8_t *data, uint16_t *par, int len, uint16_t *s, int no_eras, int *eras_pos, uint16_t invmsk, uint16_t *corr)
Decode codeword (8bit data width)

Parameters

struct rs_control *rsc
the rs control structure

uint8_t *data
data field of a given type

uint16_t *par
received parity data field

int len
data length

uint16_t *s
syndrome data field, must be in index form (if NULL, syndrome is calculated)

int no_eras
number of erasures

int *eras_pos
position of erasures, can be NULL

uint16_t invmsk
invert data mask (will be xored on data, not on parity!)

uint16_t *corr
buffer to store correction bitmask on eras_pos

The syndrome and parity uses a uint16_t data type to enable symbol size > 8. The calling code must take care of decoding of the syndrome result and the received parity before calling this code.

Note

The rs_control struct rsc contains buffers which are used for
decoding, so the caller has to ensure that decoder invocations are serialized.

Returns the number of corrected symbols or -EBADMSG for uncorrectable errors. The count includes errors in the parity.

int **encode_rs16**(struct *rs_control* *rsc, uint16_t *data, int len, uint16_t *par, uint16_t invmsk)

Calculate the parity for data values (16bit data width)

Parameters

struct rs_control *rsc
the rs control structure

uint16_t *data
data field of a given type

int len
data length

uint16_t *par
parity data, must be initialized by caller (usually all 0)

uint16_t invmsk
invert data mask (will be xored on data, not on parity!)

Each field in the data array contains up to symbol size bits of valid data.

int **decode_rs16**(struct *rs_control* *rsc, uint16_t *data, uint16_t *par, int len, uint16_t *s, int no_eras, int *eras_pos, uint16_t invmsk, uint16_t *corr)

Decode codeword (16bit data width)

Parameters

struct rs_control *rsc
the rs control structure

uint16_t *data
data field of a given type

uint16_t *par
received parity data field

int len
data length

uint16_t *s
syndrome data field, must be in index form (if NULL, syndrome is calculated)

int no_eras
number of erasures

int *eras_pos
position of erasures, can be NULL

uint16_t invmsk
invert data mask (will be xored on data, not on parity!)

uint16_t *corr
buffer to store correction bitmask on eras_pos

Each field in the data array contains up to symbol size bits of valid data.

Note

The rc_control struct rsc contains buffers which are used for
decoding, so the caller has to ensure that decoder invocations are serialized.

Returns the number of corrected symbols or -EBADMSG for uncorrectable errors. The count includes errors in the parity.

8.1.6 Credits

The library code for encoding and decoding was written by Phil Karn.

```
Copyright 2002, Phil Karn, KA9Q
May be used under the terms of the GNU General Public License (GPL)
```

The wrapper functions and interfaces are written by Thomas Gleixner.

Many users have provided bugfixes, improvements and helping hands for testing. Thanks a lot.

The following people have contributed to this document:

Thomas Gleixnertglx@linutronix.de

8.2 Netlink notes for kernel developers

8.2.1 General guidance

Attribute enums

Older families often define "null" attributes and commands with value of `0` and named `unspec`. This is supported (`type: unused`) but should be avoided in new families. The `unspec` enum values are not used in practice, so just set the value of the first attribute to `1`.

Message enums

Use the same command IDs for requests and replies. This makes it easier to match them up, and we have plenty of ID space.

Use separate command IDs for notifications. This makes it easier to sort the notifications from replies (and present them to the user application via a different API than replies).

Answer requests

Older families do not reply to all of the commands, especially NEW / ADD commands. User only gets information whether the operation succeeded or not via the ACK. Try to find useful data to return. Once the command is added whether it replies with a full message or only an ACK is uAPI and cannot be changed. It's better to err on the side of replying.

Specifically NEW and ADD commands should reply with information identifying the created object such as the allocated object's ID (without having to resort to using `NLM_F_ECHO`).

NLM_F_ECHO

Make sure to pass the request info to genl_notify() to allow `NLM_F_ECHO` to take effect. This is useful for programs that need precise feedback from the kernel (for example for logging purposes).

Support dump consistency

If iterating over objects during dump may skip over objects or repeat them - make sure to report dump inconsistency with `NLM_F_DUMP_INTR`. This is usually implemented by maintaining a generation id for the structure and recording it in the `seq` member of struct netlink_callback.

8.2.2 Netlink specification

Documentation of the Netlink specification parts which are only relevant to the kernel space.

Globals

kernel-policy

Defines whether the kernel validation policy is `global` i.e. the same for all operations of the family, defined for each operation individually - `per-op`, or separately for each operation and operation type (do vs dump) - `split`. New families should use `per-op` (default) to be able to narrow down the attributes accepted by a specific command.

checks

Documentation for the `checks` sub-sections of attribute specs.

unterminated-ok

Accept strings without the null-termination (for legacy families only). Switches from the `NLA_NUL_STRING` to `NLA_STRING` policy type.

max-len

Defines max length for a binary or string attribute (corresponding to the `len` member of struct nla_policy). For string attributes terminating null character is not counted towards `max-len`.

The field may either be a literal integer value or a name of a defined constant. String types may reduce the constant by one (i.e. specify `max-len: CONST - 1`) to reserve space for the terminating character so implementations should recognize such pattern.

min-len

Similar to `max-len` but defines minimum length.

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

Q

R

S

T

Z