The kernel core API manual

Release

The kernel development community

August 18, 2018

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

The Linux Kernel API

List Management Functions

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_entry**(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_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

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

Parameters

const struct list_head * head the list to 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

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_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(*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(*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(*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(*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(*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_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_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_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(*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(*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.

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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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.

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(*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(*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(*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 `struct hlist_node` to use as temporary storage

head the head for your list.

member the name of the hlist_node within the struct.

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 is obsolete. 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 is obsolete. 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 is obsolete. 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 is obsolete. 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 Documentation/core-api/printk-formats.rst 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 **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. Used as a replacement for the obsolete simple_strtoull. Return code must be checked.

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. Used as a replacement for the obsolete simple_strtoull. 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. Used as a replacement for the obsolete 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. Used as a replacement for the obsolete simple_strtoull. 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. Used as a replacement for the obsolete simple_strtoull. 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. Used as a replacement for the obsolete simple_strtoull. 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 'Yy1Nn0', or [oO][NnFf] for "on" and "off". Otherwise it will return -EINVAL. Value pointed to by res is updated upon finding a match.

String Manipulation

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 * **strcpy**(char * *dest*, const char * *src*)
Copy a NUL terminated string

Parameters

char * dest Where to copy the string to

const char * src Where to copy the string from

char * **strncpy**(char * *dest*, const char * *src*, size_t *count*)
Copy a length-limited, C-string

Parameters

char * dest Where to copy the string to

const char * src Where to copy the string from

size_t count The maximum number of bytes to copy

Description

The result is not `NUL-terminated` if the source exceeds **count** bytes.

In the case where the length of **src** is less than that of count, the remainder of **dest** will be padded with `NUL`.

size_t **strlcpy**(char * *dest*, const char * *src*, size_t *size*)
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 size size of destination buffer

Description

Compatible with `*BSD`: the result is always a valid NUL-terminated string that fits in the buffer (unless, of course, the buffer size is zero). It does not pad out the result like *strncpy()* does.

ssize_t **strscpy**(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 routine returns the number of characters copied (not including the trailing NUL) or -E2BIG if the destination buffer wasn't big enough. The behavior is undefined if the string buffers overlap. The destination buffer is always NUL terminated, unless it's zero-sized.

Preferred to *strlcpy()* since the API doesn't require reading memory from the src string beyond the specified "count" 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 zeroed. If the zeroing is desired, it's likely cleaner to use *strscpy()* with an overflow test, then just *memset()* the tail of the dest buffer.

char * **strcat**(char * *dest*, const char * *src*)
Append one `NUL-terminated` string to another

Parameters

char * dest The string to be appended to

const char * src The string to append to it

char * **strncat**(char * *dest*, const char * *src*, size_t *count*)
Append a length-limited, C-string to another

Parameters

char * dest The string to be appended to

const char * src The string to append to it

size_t count The maximum numbers of bytes to copy

Description

Note that in contrast to *strncpy()*, *strncat()* ensures the result is terminated.

size_t **strlcat**(char * *dest*, const char * *src*, size_t *count*)
Append a length-limited, C-string to another

Parameters

char * dest The string to be appended to

const char * src The string to append to it

size_t count The size of the destination buffer.

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

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

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**.

size_t **strlen**(const char * *s*)
Find the length of a string

Parameters

const char * s The string to be sized

size_t **strnlen**(const char * *s*, size_t *count*)
Find the length of a length-limited string

Parameters

const char * s The string to be sized

size_t count The maximum number of bytes to search

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. ;)

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

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.

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 **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.

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

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.

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**.

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

Parameters

char * s The string to operate on.

char old The character being replaced.

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

Description

Returns pointer to the nul byte at the end of **s**.

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 function is atomic and may not be reordered. See *__set_bit()* if you do not require the atomic guarantees.

Note

there are no guarantees that this function will not be reordered on non x86 architectures, so if you are writing portable code, make sure not to rely on its reordering guarantees.

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

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

Parameters

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 and may be reordered. If it's called on the same region of memory simultaneously, the effect may be that only one operation succeeds.

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

clear_bit() is atomic and may not be reordered. However, it does not contain a memory barrier, so if it is used for locking purposes, you should call `smp_mb__before_atomic()` and/or `smp_mb__after_atomic()` in order to ensure changes are visible on other processors.

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

Parameters

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 and may be reordered. If it's called on the same region of memory simultaneously, the effect may be that only one operation succeeds.

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

change_bit() is atomic and may not be reordered. 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 operation is atomic and cannot be reordered. It also implies a memory barrier.

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 is the same as test_and_set_bit on x86.

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 operation is non-atomic and can be reordered. If two examples of this operation race, one can appear to succeed but actually fail. You must protect multiple accesses with a lock.

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 operation is atomic and cannot be reordered. It also implies a 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 operation is non-atomic and can be reordered. If two examples of this operation race, one can appear to succeed but actually fail. You must protect multiple accesses with a lock.

Note

the operation is performed atomically with respect to the local CPU, but not other CPUs. Portable code should not rely on this behaviour. KVM relies on this behaviour on x86 for modifying memory that is also accessed from a hypervisor on the same CPU if running in a VM: don't change this without also updating arch/x86/kernel/kvm.c

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 operation is atomic and cannot be reordered. It also implies a memory barrier.

bool **test_bit**(int *nr*, const volatile unsigned long * *addr*)
Determine whether a bit is set

Parameters

int nr bit number to test

const volatile unsigned long * addr Address to start counting from

unsigned long **__ffs**(unsigned long *word*)
find first set bit in word

Parameters

unsigned long word The word to search

Description

Undefined if no bit exists, so code should check against 0 first.

unsigned long **ffz**(unsigned long *word*)
find first zero bit in word

Parameters

unsigned long word The word to search

Description

Undefined if no zero exists, so code should check against ~0UL first.

int **ffs**(int *x*)
find first set bit in word

Parameters

int x the word to search

Description

This is defined the same way as the libc and compiler builtin ffs routines, therefore differs in spirit from the other bitops.

ffs(value) returns 0 if value is 0 or the position of the first set bit if value is nonzero. The first (least significant) bit is at position 1.

int **fls**(int *x*)
find last set bit in word

Parameters

int x the word to search

Description

This is defined in a similar way as the libc and compiler builtin ffs, but returns the position of the most significant set bit.

fls(value) returns 0 if value is 0 or the position of the last set bit if value is nonzero. The last (most significant) bit is at position 32.

int **fls64**(__u64 *x*)
find last set bit in a 64-bit word

Parameters

__u64 x the word to search

Description

This is defined in a similar way as the libc and compiler builtin ffsll, but returns the position of the most significant set bit.

fls64(value) returns 0 if value is 0 or the position of the last set bit if value is nonzero. The last (most significant) bit is at position 64.

Bitmap Operations

bitmaps provide an array of bits, implemented using an 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.

These operations actually hold to a slightly stronger rule: if you don't input any bitmaps to these ops that have some unused bits set, then they won't output any set unused bits in output bitmaps.

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:

Note that nbits should be always a compile time evaluable constant. Otherwise many inlines will generate horrible code.

```
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_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)  as above
```

```
bitmap_shift_right(dst, src, n, nbits)      *dst = *src >> n
bitmap_shift_left(dst, src, n, nbits)       *dst = *src << n
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_to_arr32(buf, src, nbits)            Copy nbits from buf to u32[] dst
```

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.

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

convert an ASCII hex string into a bitmap.

Parameters

const char * buf 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.

int is_user location of buffer, 0 indicates kernel space

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 and for grouping errors such as "1,,5", ",44", "," and "". Leading and trailing whitespace accepted, but not embedded whitespace.

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.

Description

Wrapper for *__bitmap_parse()*, providing it with user buffer.

We cannot have this as an inline function in bitmap.h because it needs linux/uaccess.h to get the *access_ok()* declaration and this causes cyclic dependencies.

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 area and that sufficient storage remains at **buf** to accommodate the *bitmap_print_to_pagebuf()* output.

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

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.

We cannot have this as an inline function in bitmap.h because it needs linux/uaccess.h to get the *access_ok()* declaration and this causes cyclic dependencies.

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.

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.

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**

int **__bitmap_parselist**(const char * *buf*, unsigned int *buflen*, int *is_user*, unsigned long * *maskp*, int *nmaskbits*)
convert list format ASCII string to bitmap

Parameters

const char * buf read nul-terminated user string from this buffer

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

int is_user location of buffer, 0 indicates kernel space

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

Return

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

- `-EINVAL`: second number in range smaller than first
- `-EINVAL`: invalid character in string
- `-ERANGE`: bit number specified too large for mask

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**.

unsigned int **bitmap_ord_to_pos**(const unsigned long * *buf*, unsigned int *ord*, unsigned int *nbits*)
find position of n-th set bit in bitmap

Parameters

const unsigned long * buf pointer to bitmap

unsigned int ord ordinal bit position (n-th set bit, n >= 0)

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

Description

Map the ordinal offset of bit **ord** in **buf** to its position in **buf**. Value of **ord** should be in range 0 <= **ord** < weight(buf). If **ord** >= weight(buf), returns **nbits**.

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

The bit positions 0 through **nbits**-1 are valid positions in **buf**.

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.

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.

Command-line Parsing

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

Parameters

char ** str option string

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

Description

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

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

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

Description

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.

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

Description

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

Sorting

void **sort**(void * *base*, size_t *num*, size_t *size*, int (*cmp_func) (const void *, const void *, void (*swap_func) (void *, void *, int *size*)

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

int (*)(const void *, const void *) cmp_func pointer to comparison function

void (*)(void *, void *, int size) swap_func pointer to swap function or NULL

Description

This function does a heapsort on the given array. You may provide a swap_func function optimized to your element type.

Sorting time is O(n log n) both on average and worst-case. While qsort is about 20% 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*, int (*cmp) (void **priv*, struct list_head **a*, struct list_head **b*)

sort a list

Parameters

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

struct list_head * head the list to sort

int (*)(void *priv, struct list_head *a, struct list_head *b) cmp the elements comparison function

Description

This function implements "merge sort", which has O(nlog(n)) complexity.

The comparison function **cmp** must return a negative value if **a** should sort before **b**, and a positive value if **a** should sort after **b**. If **a** and **b** are equivalent, and their original relative ordering is to be preserved, **cmp** must return 0.

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 :c:func:`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 :c:func:`textsearch_find()` or
    :c:func:`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 :c:func:`find()`
    request to the algorithm.
(5) Algorithm calls :c:func:`get_next_block()` provided by the user continuously
    to fetch the data to be searched in block by block.
(6) Algorithm invokes :c:func:`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 :c:func:`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.

Note

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

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.

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

CRC and Math Functions in Linux

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*, u8 *polynomial*)
fill crc table for given polynomial in reverse bit order.

Parameters

u8 table table to be filled.

u8 polynomial polynomial for which table is to be filled.

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

Parameters

u8 table table to be filled.

u8 polynomial polynomial for which table is to be filled.

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

Parameters

const u8 table crc table used for calculation.

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*, 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 little-endian Ethernet table

u32 polynomial CRC32/CRC32c LE polynomial

u32 __attribute_const__ **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*, 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 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(*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(*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(*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(*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(*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.

Division Functions

do_div(*n*, *base*)
returns 2 values: calculate remainder and update new dividend

Parameters

n pointer to 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

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.

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

Parameters

s64 dividend signed 64bit dividend

s32 divisor signed 32bit divisor

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

Parameters

s64 dividend 64bit dividend

s32 divisor 64bit divisor

s32 * remainder 64bit remainder

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

Parameters

u64 dividend 64bit dividend

u64 divisor 64bit divisor

u64 * remainder 64bit remainder

Description

This implementation is a comparable to algorithm used by div64_u64. But this operation, which includes math for calculating the remainder, is kept distinct to avoid slowing down the div64_u64 operation on 32bit systems.

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

Parameters

u64 dividend 64bit dividend

u64 divisor 64bit divisor

Description

This implementation is a modified version of the algorithm proposed by the book 'Hacker's Delight'. The original source and full proof can be found here and is available for use without restriction.

'http://www.hackersdelight.org/hdcodetxt/divDouble.c.txt'

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

Parameters

s64 dividend 64bit dividend

s64 divisor 64bit divisor

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*)
generate a random UUID

Parameters

unsigned char uuid 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.

Memory Management in Linux

The Slab Cache

void * **kmalloc**(size_t *size*, gfp_t *flags*)
allocate memory

Parameters

size_t size how many bytes of memory are required.

gfp_t flags the type of memory to allocate.

Description

kmalloc is the normal method of allocating memory for objects smaller than page size in the kernel.

The **flags** argument may be one of:

GFP_USER - Allocate memory on behalf of user. May sleep.

GFP_KERNEL - Allocate normal kernel ram. May sleep.

GFP_ATOMIC - Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.

GFP_HIGHUSER - Allocate pages from high memory.

GFP_NOIO - Do not do any I/O at all while trying to get memory.

GFP_NOFS - Do not make any fs calls while trying to get memory.

GFP_NOWAIT - Allocation will not sleep.

__GFP_THISNODE - Allocate node-local memory only.

GFP_DMA - Allocation suitable for DMA. Should only be used for *kmalloc()* caches. Otherwise, use a slab created with SLAB_DMA.

Also it is possible to set different flags by OR'ing in one or more of the following additional **flags**:

`__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.

There are other flags available as well, but these are not intended for general use, and so are not documented here. For a full list of potential flags, always refer to linux/gfp.h.

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 * **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

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. The flags are only relevant if the cache has no available objects.

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.

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.

void **kfree**(const void * *objp*)
free previously allocated memory

Parameters

const void * objp pointer returned by kmalloc.

Description

If **objp** is NULL, no operation is performed.

Don't free memory not originally allocated by *kmalloc()* or you will run into trouble.

size_t **ksize**(const void * *objp*)
get the actual amount of memory allocated for a given object

Parameters

const void * objp Pointer to the object

Description

kmalloc may internally round up allocations and return more memory than requested. *ksize()* can be used to determine the actual amount of memory allocated. The caller may use this additional memory, even though a smaller amount of memory was initially specified with the kmalloc call. The caller must guarantee that objp points to a valid object previously allocated with either *kmalloc()* or *kmem_cache_alloc()*. The object must not be freed during the duration of the call.

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.

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

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

Description

Function returns source string if it is in .rodata section otherwise it fallbacks to kstrdup. Strings allocated by kstrdup_const should be freed by kfree_const.

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.

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

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

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

Description

Returns 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

Description

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

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

Description

Returns an ERR_PTR() on failure.

int **get_user_pages_fast**(unsigned long *start*, int *nr_pages*, int *write*, 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

int write whether pages will be written to

struct page ** pages array that receives pointers to the pages pinned. Should be at least nr_pages long.

Description

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.

get_user_pages_fast provides equivalent functionality to get_user_pages, operating on current and current->mm, with force=0 and vma=NULL. However unlike get_user_pages, it must be called without mmap_sem held.

get_user_pages_fast may take mmap_sem and page table locks, so no assumptions can be made about lack of locking. get_user_pages_fast is to be implemented in a way that is advantageous (vs get_user_pages()) when the user memory area is already faulted in and present in ptes. However if the pages have to be faulted in, it may turn out to be slightly slower so callers need to carefully consider what to use. On many architectures, get_user_pages_fast simply falls back to get_user_pages.

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.

Reclaim modifiers - __GFP_NORETRY and __GFP_NOFAIL are not supported. __GFP_RETRY_MAYFAIL is supported, and it should be used only if kmalloc is preferable to the vmalloc fallback, due to visible performance drawbacks.

Please note that any use of gfp flags outside of GFP_KERNEL is careful to not fall back to vmalloc.

User Space Memory Access

access_ok(*type*, *addr*, *size*)
Checks if a user space pointer is valid

Parameters

type Type of access: `VERIFY_READ` or `VERIFY_WRITE`. Note that `VERIFY_WRITE` is a superset of `VERIFY_READ` - if it is safe to write to a block, it is always safe to read from it.

addr User space pointer to start of block to check

size Size of block to check

Context

User context only. This function may sleep if pagefaults are enabled.

Description

Checks if a pointer to a block of memory in user space is valid.

Returns true (nonzero) if the memory block may be valid, false (zero) if it is definitely invalid.

Note that, depending on architecture, this function probably just checks that the pointer is in the user space range - after calling this function, memory access functions may still return -EFAULT.

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.

Returns zero on success, or -EFAULT on error. On error, the variable **x** is set to zero.

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**.

Returns zero on success, or -EFAULT on error.

__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.

Returns zero on success, or -EFAULT on error. On error, the variable **x** is set to zero.

__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.

Returns 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.

Returns 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.

Returns number of bytes that could not be cleared. On success, this will be zero.

More Memory Management Functions

int **read_cache_pages** (struct address_space * *mapping*, struct list_head * *pages*, int (*filler) (void *, struct page *, void * *data*)
populate an address space with some pages & start reads against them

Parameters

struct address_space * mapping the address_space

struct list_head * pages The address of a list_head which contains the target pages. These pages have their ->index populated and are otherwise uninitialised.

int (*)(void *, struct page *) filler callback routine for filling a single page.

void * data private data for the callback routine.

Description

Hides the details of the LRU cache etc from the filesystems.

void **page_cache_sync_readahead** (struct address_space * *mapping*, struct file_ra_state * *ra*, struct file * *filp*, pgoff_t *offset*, unsigned long *req_size*)
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 * filp passed on to ->:c:func:*readpage()* and ->:c:func:*readpages()*

pgoff_t offset start offset into **mapping**, in pagecache page-sized units

unsigned long req_size hint: total size of the read which the caller is performing in pagecache pages

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 * *filp*, struct page * *page*, pgoff_t *offset*, unsigned long *req_size*)
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 * filp passed on to ->:c:func:*readpage()* and ->:c:func:*readpages()*

struct page * page the page at **offset** which has the PG_readahead flag set

pgoff_t offset start offset into **mapping**, in pagecache page-sized units

unsigned long req_size hint: total size of the read which the caller is performing in pagecache pages

Description

page_cache_async_readahead() should be called when a page is used which has the PG_readahead flag; 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.

void **delete_from_page_cache**(struct page * *page*)
delete page from page cache

Parameters

struct page * page the page which the kernel is trying to remove from page cache

Description

This must be called only on pages that have been verified to be in the page cache and locked. It will never put the page into the free list, the caller has a reference on the page.

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.

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.

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.

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.

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)

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).

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.

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.

int **replace_page_cache_page**(struct page * *old*, struct page * *new*, gfp_t *gfp_mask*)
replace a pagecache page with a new one

Parameters

struct page * old page to be replaced

struct page * new page to replace with

gfp_t gfp_mask allocation mode

Description

This function replaces a page in the pagecache with a new one. On success it acquires the pagecache reference for the new page and drops it for the old page. Both the old and new pages must be locked. This function does not add the new page to the LRU, the caller must do that.

The remove + add is atomic. The only way this function can fail is memory allocation failure.

int **add_to_page_cache_locked**(struct page * *page*, struct address_space * *mapping*, pgoff_t *offset*, gfp_t *gfp_mask*)
add a locked page to the pagecache

Parameters

struct page * page page to add

struct address_space * mapping the page's address_space

pgoff_t offset page index

gfp_t gfp_mask page allocation mode

Description

This function is used to add a page to the pagecache. It must be locked. This function does not add the page to the LRU. The caller must do that.

void **add_page_wait_queue**(struct page * *page*, wait_queue_entry_t * *waiter*)
Add an arbitrary waiter to a page's wait queue

Parameters

struct page * page Page 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 **page**.

void **unlock_page**(struct page * *page*)
unlock a locked page

Parameters

struct page * page the page

Description

Unlocks the page and wakes up sleepers in `___wait_on_page_locked()`. Also wakes sleepers in `wait_on_page_writeback()` because the wakeup mechanism between PageLocked pages and PageWriteback pages is shared. But that's OK - sleepers in `wait_on_page_writeback()` just go back to sleep.

Note that this depends on PG_waiters being the sign bit in the byte that contains PG_locked - thus the `BUILD_BUG_ON()`. That allows us to clear the PG_locked bit and test PG_waiters at the same time fairly portably (architectures that do LL/SC can test any bit, while x86 can test the sign bit).

void **end_page_writeback**(struct page * *page*)
end writeback against a page

Parameters

struct page * page the page

void **__lock_page**(struct page * *__page*)
get a lock on the page, assuming we need to sleep to get it

Parameters

struct page * __page the page to lock

pgoff_t **page_cache_next_hole**(struct address_space * *mapping*, pgoff_t *index*, unsigned long *max_scan*)
find the next hole (not-present entry)

Parameters

struct address_space * mapping mapping

pgoff_t index index

unsigned long max_scan maximum range to search

Description

Search the set [index, min(index+max_scan-1, MAX_INDEX)] for the lowest indexed hole.

Return

the index of the hole if found, otherwise returns an index outside of the set specified (in which case 'return - index >= max_scan' will be true). In rare cases of index wrap-around, 0 will be returned.

page_cache_next_hole may be called under rcu_read_lock. However, like radix_tree_gang_lookup, this will not atomically search a snapshot of the tree at a single point in time. For example, if a hole is created at index 5, then subsequently a hole is created at index 10, page_cache_next_hole covering both indexes may return 10 if called under rcu_read_lock.

pgoff_t **page_cache_prev_hole**(struct address_space * *mapping*, pgoff_t *index*, unsigned long *max_scan*)
find the prev hole (not-present entry)

Parameters

struct address_space * mapping mapping

pgoff_t index index

unsigned long max_scan maximum range to search

Description

Search backwards in the range [max(index-max_scan+1, 0), index] for the first hole.

Return

the index of the hole if found, otherwise returns an index outside of the set specified (in which case 'index - return >= max_scan' will be true). In rare cases of wrap-around, ULONG_MAX will be returned.

page_cache_prev_hole may be called under rcu_read_lock. However, like radix_tree_gang_lookup, this will not atomically search a snapshot of the tree at a single point in time. For example, if a hole is created at index 10, then subsequently a hole is created at index 5, page_cache_prev_hole covering both indexes may return 5 if called under rcu_read_lock.

struct page * **find_get_entry**(struct address_space * *mapping*, pgoff_t *offset*)
find and get a page cache entry

Parameters

struct address_space * mapping the address_space to search

pgoff_t offset the page cache 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.

If the slot holds a shadow entry of a previously evicted page, or a swap entry from shmem/tmpfs, it is returned.

Otherwise, NULL is returned.

struct page * **find_lock_entry**(struct address_space * *mapping*, pgoff_t *offset*)
locate, pin and lock a page cache entry

Parameters

struct address_space * mapping the address_space to search

pgoff_t offset the page cache index

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 slot holds a shadow entry of a previously evicted page, or a swap entry from shmem/tmpfs, it is returned.

Otherwise, NULL is returned.

find_lock_entry() may sleep.

struct page * **pagecache_get_page**(struct address_space * *mapping*, pgoff_t *offset*, int *fgp_flags*, gfp_t *gfp_mask*)
find and get a page reference

Parameters

struct address_space * mapping the address_space to search

pgoff_t offset the page index

int fgp_flags PCG flags

gfp_t gfp_mask gfp mask to use for the page cache data page allocation

Description

Looks up the page cache slot at **mapping** & **offset**.

PCG flags modify how the page is returned.

fgp_flags can be:

- FGP_ACCESSED: the page will be marked accessed
- FGP_LOCK: Page is return locked

- FGP_CREAT: If page is not present then 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. Otherwise, NULL is returned.

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 there is a page cache page, it is returned with an increased refcount.

unsigned **`find_get_pages_contig`**(struct address_space * *mapping*, pgoff_t *index*, unsigned int *nr_pages*, struct page ** *pages*)
gang contiguous pagecache lookup

Parameters

`struct address_space * mapping` The address_space to search

`pgoff_t index` The starting page index

`unsigned int nr_pages` The maximum number of pages

`struct page ** pages` Where the resulting pages are placed

Description

`find_get_pages_contig()` works exactly like `find_get_pages()`, except that the returned number of pages are guaranteed to be contiguous.

`find_get_pages_contig()` returns the number of pages which were found.

unsigned **`find_get_pages_range_tag`**(struct address_space * *mapping*, pgoff_t * *index*, pgoff_t *end*, int *tag*, unsigned int *nr_pages*, struct page ** *pages*)
find and return pages in given range matching **tag**

Parameters

`struct address_space * mapping` the address_space to search

`pgoff_t * index` the starting page index

`pgoff_t end` The final page index (inclusive)

`int tag` the tag index

`unsigned int nr_pages` the maximum number of pages

`struct page ** pages` where the resulting pages are placed

Description

Like find_get_pages, except we only return pages which are tagged with **tag**. We update **index** to index the next page for the traversal.

unsigned **`find_get_entries_tag`**(struct address_space * *mapping*, pgoff_t *start*, int *tag*, unsigned int *nr_entries*, struct page ** *entries*, pgoff_t * *indices*)
find and return entries that match **tag**

Parameters

`struct address_space * mapping` the address_space to search

`pgoff_t start` the starting page cache index

`int tag` the tag index

`unsigned int nr_entries` the maximum number of entries

`struct page ** entries` where the resulting entries are placed

`pgoff_t * indices` the cache indices corresponding to the entries in **entries**

Description

Like find_get_entries, except we only return entries which are tagged with **tag**.

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.

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_sem must be held on entry.

If our return value has VM_FAULT_RETRY set, it's because `lock_page_or_retry()` returned 0. The mmap_sem has usually been released in this case. See `__lock_page_or_retry()` for the exception.

If our return value does not have VM_FAULT_RETRY set, the mmap_sem has not been released.

We never return with VM_FAULT_RETRY and a bit from VM_FAULT_ERROR set.

struct page * **read_cache_page**(struct address_space * *mapping*, pgoff_t *index*, int (*filler) (void *, struct page *, void * *data*)
read into page cache, fill it if needed

Parameters

struct address_space * mapping the page's address_space

pgoff_t index the page index

int (*)(void *, struct page *) filler function to perform the read

void * data first arg to filler(data, page) function, often left as NULL

Description

Read into the page cache. If a page already exists, and `PageUptodate()` is not set, try to fill the page and wait for it to become unlocked.

If the page does not get brought uptodate, return -EIO.

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.

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_mutex 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_mutex.

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_mutex as needed.

int **try_to_release_page**(struct page * *page*, gfp_t *gfp_mask*)
release old fs-specific metadata on a page

Parameters

struct page * page the page which the kernel is trying to free

gfp_t gfp_mask memory allocation flags (and I/O mode)

Description

The address_space is to try to release any data against the page (presumably at page->private). If the release was successful, return ‘1’. Otherwise return zero.

This may also be called if PG_fscache is set on a page, indicating that the page is known to the local caching routines.

The **gfp_mask** argument specifies whether I/O may be performed to release this page (__GFP_IO), and whether the call may block (__GFP_RECLAIM & __GFP_FS).

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_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->:c:func:*mmap()* handler under mm->mmap_sem 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.

int **vm_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 NULL.

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.

int **vm_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 vm_insert_pfn, except that it allows drivers to 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; vm_insert_pfn_prot should only be used if using multiple VMAs is impractical.

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 user address to start at

unsigned long pfn physical address of kernel memory

unsigned long size size of map area

pgprot_t prot page protection flags for this mapping

Note

this is only safe if the mm semaphore is held when called.

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 area

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.

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_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.

Returns zero and the pfn at **pfn** on success, -ve otherwise.

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*, pgprot_t *prot*)
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

pgprot_t prot memory protection to use. PAGE_KERNEL for regular RAM

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 **unmap_kernel_range_noflush**(unsigned long *addr*, unsigned long *size*)
unmap kernel VM area

Parameters

unsigned long addr start of the VM area to unmap

unsigned long size size of the VM area to unmap

Description

Unmap PFN_UP(**size**) pages at **addr**. The VM area **addr** and **size** specify should have been allocated using `get_vm_area()` and its friends.

NOTE

This function does NOT do any cache flushing. The caller is responsible for calling `flush_cache_vunmap()` on to-be-mapped areas before calling this function and `flush_tlb_kernel_range()` after.

void **unmap_kernel_range**(unsigned long *addr*, unsigned long *size*)
unmap kernel VM area and flush cache and TLB

Parameters

unsigned long addr start of the VM area to unmap

unsigned long size size of the VM area to unmap

Description

Similar to *unmap_kernel_range_noflush()* but flushes vcache before the unmapping and tlb after.

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 *vmalloc()*, *vmalloc_32()* or `__vmalloc()`. If **addr** is NULL, no operation is performed.

Must not be called in NMI context (strictly speaking, only if we don't have CONFIG_ARCH_HAVE_NMI_SAFE_CMPXCHG, but making the calling conventions for *vfree()* arch-depenedent would be a really bad idea)

NOTE

assumes that the object at **addr** has a size >= sizeof(llist_node)

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.

void * **vmalloc**(unsigned long *size*)
allocate virtually contiguous memory

Parameters

unsigned long size allocation size Allocate enough pages to cover **size** from the page level allocator and map them into contiguous kernel virtual space.

Description

For tight control over page level allocator and protection flags use `__vmalloc()` instead.

void * **vzalloc**(unsigned long *size*)
allocate virtually contiguous memory with zero fill

Parameters

unsigned long size allocation size 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.

Description

For tight control over page level allocator and protection flags use `__vmalloc()` instead.

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.

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.

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.

For tight control over page level allocator and protection flags use `__vmalloc_node()` instead.

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.

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.

int **remap_vmalloc_range_partial**(struct vm_area_struct * *vma*, unsigned long *uaddr*, void * *kaddr*, unsigned long *size*)
map vmalloc pages to userspace

Parameters

struct vm_area_struct * vma vma to cover

unsigned long uaddr target user address to start at

void * kaddr virtual address of vmalloc kernel memory

unsigned long size size of map area

Return

0 for success, -Exxx on failure

This function checks that **kaddr** is a valid vmalloc'ed area, and that it is big enough to cover the range starting at **uaddr** in **vma**. Will return failure if that criteria isn't met.

Similar to *remap_pfn_range()* (see mm/memory.c)

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

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)

struct vm_struct * **alloc_vm_area**(size_t *size*, pte_t ** *ptes*)
allocate a range of kernel address space

Parameters

size_t size size of the area

pte_t ** ptes returns the PTEs for the address space

Return

NULL on failure, vm_struct on success

This function reserves a range of kernel address space, and allocates pagetables to map that range. No actual mappings are created.

If **ptes** is non-NULL, pointers to the PTEs (in init_mm) allocated for the VM area are returned.

unsigned long **__get_pfnblock_flags_mask**(struct page * *page*, unsigned long *pfn*, unsigned long *end_bitidx*, unsigned long *mask*)

Return the requested group of flags for the pageblock_nr_pages block of pages

Parameters

struct page * page The page within the block of interest

unsigned long pfn The target page frame number

unsigned long end_bitidx The last bit of interest to retrieve

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 *end_bitidx*, 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 end_bitidx The last bit of interest

unsigned long mask mask of bits that the caller is interested in

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

Description

Like `alloc_pages_exact()`, but try to allocate on node nid first before falling back.

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 counts 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

unsigned long **nr_free_pagecache_pages**(void)

count number of pages beyond high watermark

Parameters

void no arguments

Description

nr_free_pagecache_pages() counts the number of pages which are beyond the high watermark within all zones.

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. It returns -1 if no node is found.

void **free_bootmem_with_active_regions**(int *nid*, unsigned long *max_low_pfn*)
Call memblock_free_early_nid for each active range

Parameters

int nid The node to free memory on. If MAX_NUMNODES, all nodes are freed.

unsigned long max_low_pfn The highest PFN that will be passed to memblock_free_early_nid

Description

If an architecture guarantees that all ranges registered contain no holes and may be freed, this this function may be used instead of calling `memblock_free_early_nid()` manually.

void **sparse_memory_present_with_active_regions**(int *nid*)
Call memory_present for each active range

Parameters

int nid The node to call memory_present for. If MAX_NUMNODES, all nodes will be used.

Description

If an architecture guarantees that all ranges registered contain no holes and may be freed, this function may be used instead of calling `memory_present()` manually.

void **get_pfn_range_for_nid**(unsigned int *nid*, unsigned long * *start_pfn*, unsigned long * *end_pfn*)
Return the start and end page frames for a node

Parameters

unsigned int nid The nid to return the range for. If MAX_NUMNODES, the min and max PFN are returned.

unsigned long * start_pfn Passed by reference. On return, it will have the node start_pfn.

unsigned long * end_pfn Passed by reference. On return, it will have the node end_pfn.

Description

It returns the start and end page frame of a node based on information provided by `memblock_set_node()`. If called for a node with no available memory, a warning is printed and the start and end PFNs will be 0.

unsigned long **absent_pages_in_range**(unsigned long *start_pfn*, unsigned long *end_pfn*)
Return number of page frames in holes within a range

Parameters

unsigned long start_pfn The start PFN to start searching for holes

unsigned long end_pfn The end PFN to stop searching for holes

Description

It returns the number of pages frames in memory holes within a range.

unsigned long **node_map_pfn_alignment**(void)
determine the maximum internode alignment

Parameters

void no arguments

Description

This function should be called after node map is populated and sorted. It calculates the maximum power of two alignment which can distinguish all the nodes.

For example, if all nodes are 1GiB and aligned to 1GiB, the return value would indicate 1GiB alignment with (1 << (30 - PAGE_SHIFT)). If the nodes are shifted by 256MiB, 256MiB. Note that if only the last node is shifted, 1GiB is enough and this function will indicate so.

This is used to test whether pfn -> nid mapping of the chosen memory model has fine enough granularity to avoid incorrect mapping for the populated node map.

Returns the determined alignment in pfn's. 0 if there is no alignment requirement (single node).

unsigned long **find_min_pfn_with_active_regions**(void)
Find the minimum PFN registered

Parameters

void no arguments

Description

It returns the minimum PFN based on information provided via `memblock_set_node()`.

void **free_area_init_nodes**(unsigned long * *max_zone_pfn*)
Initialise all pg_data_t and zone data

Parameters

unsigned long * max_zone_pfn an array of max PFNs for each zone

Description

This will call `free_area_init_node()` for each active node in the system. Using the page ranges provided by `memblock_set_node()`, the size of each zone in each node and their holes is calculated. If the maximum PFN between two adjacent zones match, it is assumed that the zone is empty. For example, if arch_max_dma_pfn == arch_max_dma32_pfn, it is assumed that arch_max_dma32_pfn has no pages. It is also assumed that a zone starts where the previous one ended. For example, ZONE_DMA32 starts at arch_max_dma_pfn.

void **set_dma_reserve**(unsigned long *new_dma_reserve*)
set the specified number of pages reserved in the first zone

Parameters

unsigned long new_dma_reserve The number of pages to mark reserved

Description

The per-cpu batchsize and zone watermarks are determined by managed_pages. In the DMA zone, a significant percentage may be consumed by kernel image and other unfreeable allocations which can skew the watermarks badly. This function may optionally be used to account for unfreeable pages in the first zone (e.g., ZONE_DMA). The effect will be lower watermarks and smaller per-cpu batchsize.

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 underlaying 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 or MAX_ORDER_NR_PAGES 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.

Returns 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()`.

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 *undescribed*

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).

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.

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.

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.

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.

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

!:c:func:*in_interrupt()*

Description

Returns a dma allocation pool with the requested characteristics, or null if one can't be created. 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.

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

!:c:func:*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

Description

This returns 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.

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()*.

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.

On really big machines, get_writeback_state is expensive, so try to avoid calling it too often (ratelimiting). But 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).

int **generic_writepages**(struct address_space * *mapping*, struct writeback_control * *wbc*)
walk the list of dirty pages of the given address space and `writepage()` 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

Description

This is a library function, which implements the `writepages()` address_space_operation.

int **write_one_page**(struct page * *page*)
write out a single page and wait on I/O

Parameters

struct page * page the page to write

Description

The page must be locked by the caller and will be unlocked upon return.

Note that the mapping's AS_EIO/AS_ENOSPC flags will be cleared when this function returns.

void **wait_for_stable_page**(struct page * *page*)
wait for writeback to finish, if necessary.

Parameters

struct page * page The page to wait on.

Description

This function determines if the given page is related to a backing device that requires page contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete.

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 ->:c:func:*invalidatepage()* 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_mutex.

Note

When this function returns, there can be a page in the process of deletion (inside `__delete_from_page_cache()`) 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_mutex.

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 the unlocked pages of one inode

Parameters

struct address_space * mapping the address_space which holds the pages to invalidate

pgoff_t start the offset 'from' which to invalidate

pgoff_t end the offset 'to' which to invalidate (inclusive)

Description

This function only removes the unlocked pages, if you want to remove all the pages of one inode, you must call truncate_inode_pages.

invalidate_mapping_pages() will not block on IO activity. It will not invalidate pages which are dirty, locked, under writeback or mapped into pagetables.

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.

Returns -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.

Returns -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_mutex 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_mutex - 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.

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.

int **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 IPCMNI) 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 (*)(struct seq_file *, void *) show 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 id 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.

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.

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_lock**(struct ipc_ids * *ids*, int *id*)
lock an ipc structure without rwsem held

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 lock the associated ipc object.

The ipc object is locked on successful 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 reference counter.

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_*:c:func:*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_pre_down_nolock**(struct ipc_namespace * *ns*, struct ipc_ids * *ids*, int *id*, int *cmd*, struct ipc64_perm * *perm*, int *extra_perm*)
retrieve an ipc and check permissions for some IPC_XXX cmd

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 must be called without any lock held and:

- retrieves the ipc 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.

FIFO Buffer

kfifo interface

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(*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(*fifo*)
Initialize a fifo declared by DECLARE_KFIFO

Parameters

fifo name of the declared fifo datatype

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(*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(*fifo*)
returns the size of the element managed by the fifo

Parameters

fifo address of the fifo to be used

kfifo_recsize(*fifo*)
returns the size of the record length field

Parameters

fifo address of the fifo to be used

kfifo_size(*fifo*)
returns the size of the fifo in elements

Parameters

fifo address of the fifo to be used

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(*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(*fifo*)
returns the number of used elements in the fifo

Parameters

fifo address of the fifo to be used

kfifo_is_empty(*fifo*)
returns true if the fifo is empty

Parameters

fifo address of the fifo to be used

kfifo_is_full(*fifo*)
returns true if the fifo is full

Parameters

fifo address of the fifo to be used

kfifo_avail(*fifo*)
returns the number of unused elements in the fifo

Parameters

fifo address of the fifo to be used

kfifo_skip(*fifo*)
skip output data

Parameters

fifo address of the fifo to be used

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(*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(*fifo*)
frees the fifo

Parameters

fifo the fifo to be freed

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(*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(*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(*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(*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(*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_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(*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_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(*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(*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(*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(*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(*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(*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.

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

Description

Returns 1 if the buffer is full, 0 otherwise.

void **relay_reset**(struct rchan * *chan*)
reset the channel

Parameters

struct rchan * chan the channel

Description

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*, 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

struct rchan_callbacks * cb client callback functions

void * private_data user-defined data

Description

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

Description

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

Description

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

Description

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

Description

Closes all channel buffers and frees the channel.

void **relay_flush**(struct rchan * *chan*)
close the channel

Parameters

struct rchan * chan the channel

Description

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

Description

Returns 0 if ok, negative on error

Caller should already have grabbed mmap_sem.

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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**(size_t *read_pos*, struct rchan_buf * *buf*)
find the first available byte to read

Parameters

size_t read_pos file read position

struct rchan_buf * buf relay channel buffer

Description

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

Module Support

Module 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 becomes a no-operation.

Inter Module support

Refer to the file kernel/module.c for more information.

Hardware Interfaces

Interrupt Handling

bool **synchronize_hardirq**(unsigned int *irq*)
wait for pending hard IRQ handlers (on other CPUs)

Parameters

unsigned int irq interrupt number to wait for

Description

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.

void **synchronize_irq**(unsigned int *irq*)
wait for pending IRQ handlers (on other CPUs)

Parameters

unsigned int irq interrupt number to wait for

Description

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.

This function may be called - with care - from IRQ context.

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.

Description

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

Description

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

Description

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

Description

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.

This function may be called - with care - from IRQ context.

bool **disable_hardirq**(unsigned int *irq*)
disables an irq and waits for hardirq completion

Parameters

unsigned int irq Interrupt to disable

Description

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 **enable_irq**(unsigned int *irq*)
enable handling of an irq

Parameters

unsigned int irq Interrupt to enable

Description

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 !

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

Description

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".

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

int **setup_irq**(unsigned int *irq*, struct *irqaction* * *act*)
setup an interrupt

Parameters

unsigned int irq Interrupt line to setup

struct irqaction * act irqaction for the interrupt

Description

Used to statically setup interrupts in the early boot process.

void **remove_irq**(unsigned int *irq*, struct *irqaction* * *act*)
free an 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.

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

Description

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 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

Description

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

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

Description

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.

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 **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

Description

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 **__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

Description

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 **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 storeed

Description

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**

Description

This call sets the internal irqchip state of an interrupt, depending on the value of **which**.

This function should be called with preemption disabled if the interrupt controller has per-cpu registers.

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 **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.

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.

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.

int **release_mem_region_adjustable**(struct resource * *parent*, resource_size_t *start*, resource_size_t *size*)
release a previously reserved memory region

Parameters

struct resource * parent parent resource descriptor

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.

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 **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 (*)(void *, const struct resource *, resource_size_t, resource_size_t) alignf
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.

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()*.

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.

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.

int **security_module_enable**(const char * *module*)
Load given security module on boot ?

Parameters

const char * module the name of the module

Description

Each LSM must pass this method before registering its own operations to avoid security registration races. This method may also be used to check if your LSM is currently loaded during kernel initialization.

Return

true if:

- The passed LSM is the one chosen by user at boot time,
- or the passed LSM is configured as the default and the user did not choose an alternate LSM at boot time.

Otherwise, return false.

void **security_add_hooks**(struct security_hook_list * *hooks*, int *count*, 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

char * lsm the name of the security module

Description

Each LSM has to register its hooks with the infrastructure.

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.

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 tasklet 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_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_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 and do_exit

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_RECORD_CONTEXT, 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_RECORD_CONTEXT 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**(const __user 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.

int **audit_set_loginuid**(kuid_t *loginuid*)
set current task's audit_context loginuid

Parameters

kuid_t loginuid loginuid value

Description

Returns 0.

Called (set) from fs/proc/base.c::`proc_loginuid_write()`.

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**(int *sig*, struct task_struct * *t*)
record signal info for shutting down audit subsystem

Parameters

int sig signal value

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 char * *dname*, const char * *path*, int *parentlen*)
compare given dentry name with last component in given path. Return of 0 indicates a match.

Parameters

const char * 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.

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

Returns 0 for success or negative errno values for failure.

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.

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)

Parameters

void no arguments

Description

handles process accounting for an exiting task

Block Devices

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.

bool **blk_queue_flag_test_and_clear**(unsigned int *flag*, struct request_queue * *q*)
atomically test and clear a queue flag

Parameters

unsigned int flag flag to be cleared

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 set.

void **blk_delay_queue**(struct request_queue * *q*, unsigned long *msecs*)
restart queueing after defined interval

Parameters

struct request_queue * q The `struct request_queue` in question

unsigned long msecs Delay in msecs

Description

Sometimes queueing needs to be postponed for a little while, to allow resources to come back. This function will make sure that queueing is restarted around the specified time.

void **blk_start_queue_async**(struct request_queue * *q*)
asynchronously restart a previously stopped queue

Parameters

struct request_queue * q The struct request_queue in question

Description

blk_start_queue_async() will clear the stop flag on the queue, and ensure that the request_fn for the queue is run from an async context.

void **blk_start_queue**(struct request_queue * *q*)
restart a previously stopped queue

Parameters

struct request_queue * q The struct request_queue in question

Description

blk_start_queue() will clear the stop flag on the queue, and call the request_fn for the queue if it was in a stopped state when entered. Also see *blk_stop_queue()*.

void **blk_stop_queue**(struct request_queue * *q*)
stop a queue

Parameters

struct request_queue * q The struct request_queue in question

Description

The Linux block layer assumes that a block driver will consume all entries on the request queue when the request_fn strategy is called. Often this will not happen, because of hardware limitations (queue depth settings). If a device driver gets a 'queue full' response, or if it simply chooses not to queue more I/O at one point, it can call this function to prevent the request_fn from being called until the driver has signalled it's ready to go again. This happens by calling *blk_start_queue()* to restart queue operations.

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 ->make_request_fn 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.

int **blk_set_preempt_only**(struct request_queue * *q*)
set QUEUE_FLAG_PREEMPT_ONLY

Parameters

struct request_queue * q request queue pointer

Description

Returns the previous value of the PREEMPT_ONLY flag - 0 if the flag was not set and 1 if the flag was already set.

void **__blk_run_queue_uncond**(struct request_queue * *q*)
run a queue whether or not it has been stopped

Parameters

struct request_queue * q The queue to run

Description

Invoke request handling on a queue if there are any pending requests. May be used to restart request handling after a request has completed. This variant runs the queue whether or not the queue has been stopped. Must be called with the queue lock held and interrupts disabled. See also **blk_run_queue**.

void **__blk_run_queue**(struct request_queue * *q*)
run a single device queue

Parameters

struct request_queue * q The queue to run

Description

See **blk_run_queue**.

void **blk_run_queue_async**(struct request_queue * *q*)
run a single device queue in workqueue context

Parameters

struct request_queue * q The queue to run

Description

Tells kblockd to perform the equivalent of **blk_run_queue** on behalf of us.

Note

Since it is not allowed to run q->delay_work after *blk_cleanup_queue()* has canceled q->delay_work, callers must hold the queue lock to avoid race conditions between *blk_cleanup_queue()* and *blk_run_queue_async()*.

void **blk_run_queue**(struct request_queue * *q*)
run a single device queue

Parameters

struct request_queue * q The queue to run

Description

Invoke request handling on this queue, if it has pending work to do. May be used to restart queueing when a request has completed.

void **blk_queue_bypass_start**(struct request_queue * *q*)
enter queue bypass mode

Parameters

struct request_queue * q queue of interest

Description

In bypass mode, only the dispatch FIFO queue of **q** is used. This function makes **q** enter bypass mode and drains all requests which were throttled or issued before. On return, it's guaranteed that no request is being throttled or has ELVPRIV set and `blk_queue_bypass()` `true` inside queue or RCU read lock.

void **blk_queue_bypass_end**(struct request_queue * *q*)
leave queue bypass mode

Parameters

struct request_queue * q queue of interest

Description

Leave bypass mode and restore the normal queueing behavior.

Note

although *blk_queue_bypass_start()* is only called for blk-sq queues, this function is called for both blk-sq and blk-mq queues.

void **blk_cleanup_queue**(struct request_queue * *q*)
shutdown a request queue

Parameters

struct request_queue * q request queue to shutdown

Description

Mark **q** DYING, drain all pending requests, mark **q** DEAD, destroy and put it. All future requests will be failed immediately with -ENODEV.

struct request_queue * **blk_alloc_queue_node**(gfp_t *gfp_mask*, int *node_id*, spinlock_t * *lock*)
allocate a request queue

Parameters

gfp_t gfp_mask memory allocation flags

int node_id NUMA node to allocate memory from

spinlock_t * lock For legacy queues, pointer to a spinlock that will be used to e.g. serialize calls to the legacy .:c:func:*request_fn()* callback. Ignored for blk-mq request queues.

Note

pass the queue lock as the third argument to this function instead of setting the queue lock pointer explicitly to avoid triggering a sporadic crash in the blkcg code. This function namely calls `blkcg_init_queue()` and the queue lock pointer must be set before `blkcg_init_queue()` is called.

struct request_queue * **blk_init_queue**(request_fn_proc * *rfn*, spinlock_t * *lock*)
prepare a request queue for use with a block device

Parameters

request_fn_proc * rfn The function to be called to process requests that have been placed on the queue.

spinlock_t * lock Request queue spin lock

Description

If a block device wishes to use the standard request handling procedures, which sorts requests and coalesces adjacent requests, then it must call *blk_init_queue()*. The function **rfn** will be called when there are requests on the queue that need to be processed. If the device supports plugging, then **rfn** may not be called immediately when requests are available on the queue, but may be called at some time later instead. Plugged queues are generally unplugged when a buffer belonging to one of the requests on the queue is needed, or due to memory pressure.

rfn is not required, or even expected, to remove all requests off the queue, but only as many as it can handle at a time. If it does leave requests on the queue, it is responsible for arranging that the requests get dealt with eventually.

The queue spin lock must be held while manipulating the requests on the request queue; this lock will be taken also from interrupt context, so irq disabling is needed for it.

Function returns a pointer to the initialized request queue, or `NULL` if it didn't succeed.

Note

blk_init_queue() must be paired with a *blk_cleanup_queue()* call when the block device is deactivated (such as at module unload).

struct request * **blk_get_request**(struct request_queue * *q*, unsigned int *op*, blk_mq_req_flags_t *flags*)
allocate a request

Parameters

struct request_queue * q request queue to allocate a request for

unsigned int op operation (REQ_OP_*) and REQ_* flags, e.g. REQ_SYNC.

blk_mq_req_flags_t flags BLK_MQ_REQ_* flags, e.g. BLK_MQ_REQ_NOWAIT.

void **blk_requeue_request**(struct request_queue * *q*, struct request * *rq*)
put a request back on queue

Parameters

struct request_queue * q request queue where request should be inserted

struct request * rq request to be inserted

Description

Drivers often keep queueing requests until the hardware cannot accept more, when that condition happens we need to put the request back on the queue. Must be called with queue lock held.

void **part_round_stats**(struct request_queue * *q*, int *cpu*, struct hd_struct * *part*)
Round off the performance stats on a struct disk_stats.

Parameters

struct request_queue * q target block queue

int cpu cpu number for stats access

struct hd_struct * part target partition

Description

The average IO queue length and utilisation statistics are maintained by observing the current state of the queue length and the amount of time it has been in this state for.

Normally, that accounting is done on IO completion, but that can result in more than a second's worth of IO being accounted for within any one second, leading to >100% utilisation. To deal with that, we call this function to do a round-off before returning the results when reading /proc/diskstats. This accounts immediately for all queue usage up to the current jiffies and restarts the counters again.

blk_qc_t **generic_make_request**(struct bio * *bio*)
hand a buffer to its device driver for I/O

Parameters

struct bio * bio The bio describing the location in memory and on the device.

Description

generic_make_request() is used to make I/O requests of block devices. It is passed a `struct bio`, which describes the I/O that needs to be done.

generic_make_request() does not return any status. The success/failure status of the request, along with notification of completion, is delivered asynchronously through the bio->bi_end_io function described (one day) else where.

The caller of generic_make_request must make sure that bi_io_vec are set to describe the memory buffer, and that bi_dev and bi_sector are set to describe the device address, and the bi_end_io and optionally bi_private are set to describe how completion notification should be signaled.

generic_make_request and the drivers it calls may use bi_next if this bio happens to be merged with someone else, and may resubmit the bio to a lower device by calling into generic_make_request recursively, which means the bio should NOT be touched after the call to ->make_request_fn.

blk_qc_t **direct_make_request**(struct bio * *bio*)
hand a buffer directly to its device driver for I/O

Parameters

struct bio * bio The bio describing the location in memory and on the device.

Description

This function behaves like *generic_make_request()*, but does not protect against recursion. Must only be used if the called driver is known to not call generic_make_request (or direct_make_request) again from its make_request function. (Calling direct_make_request again from a workqueue is perfectly fine as that doesn't recurse).

blk_qc_t **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 very similar in purpose to *generic_make_request()*, and uses that function to do most of the work. Both are fairly rough interfaces; **bio** must be presetup and ready for I/O.

blk_status_t **blk_insert_cloned_request**(struct request_queue * *q*, struct request * *rq*)
Helper for stacking drivers to submit a request

Parameters

struct request_queue * q the queue to submit the request

struct request * rq the request being queued

unsigned int **blk_rq_err_bytes**(const struct request * *rq*)
determine number of bytes till the next failure boundary

Parameters

const struct request * rq request to examine

Description

A request could be merge of IOs which require different failure handling. This function determines the number of bytes which can be failed from the beginning of the request without crossing into area which need to be retried further.

Return

The number of bytes to fail.

struct request * **blk_peek_request**(struct request_queue * *q*)
peek at the top of a request queue

Parameters

struct request_queue * q request queue to peek at

Description

Return the request at the top of **q**. The returned request should be started using *blk_start_request()* before LLD starts processing it.

Return

Pointer to the request at the top of **q** if available. Null otherwise.

void **blk_start_request**(struct request * *req*)
start request processing on the driver

Parameters

struct request * req request to dequeue

Description

Dequeue **req** and start timeout timer on it. This hands off the request to the driver.

struct request * **blk_fetch_request**(struct request_queue * *q*)
fetch a request from a request queue

Parameters

struct request_queue * q request queue to fetch a request from

Description

Return the request at the top of **q**. The request is started on return and LLD can start processing it immediately.

Return

Pointer to the request at the top of **q** if available. Null otherwise.

bool **blk_update_request**(struct request * *req*, blk_status_t *error*, unsigned int *nr_bytes*)
Special helper function for request stacking drivers

Parameters

struct request * req the request being processed

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete **req**

Description

Ends I/O on a number of bytes attached to **req**, but doesn't complete the request structure even if **req** doesn't have leftover. If **req** has leftover, sets it up for the next range of segments.

This special helper function is only for request stacking drivers (e.g. request-based dm) so that they can handle partial completion. Actual device drivers should use blk_end_request instead.

Passing the result of `blk_rq_bytes()` as **nr_bytes** guarantees `false` return from this function.

Return

`false` - this request doesn't have any more data `true` - this request has more data

void **blk_unprep_request**(struct request * *req*)
unprepare a request

Parameters

struct request * req the request

Description

This function makes a request ready for complete resubmission (or completion). It happens only after all error handling is complete, so represents the appropriate moment to deallocate any resources that were allocated to the request in the prep_rq_fn. The queue lock is held when calling this.

bool **blk_end_request**(struct request * *rq*, blk_status_t *error*, unsigned int *nr_bytes*)
Helper function for drivers to complete the request.

Parameters

struct request * rq the request being processed

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete

Description

Ends I/O on a number of bytes attached to **rq**. If **rq** has leftover, sets it up for the next range of segments.

Return

`false` - we are done with this request `true` - still buffers pending for this request

void **blk_end_request_all**(struct request * *rq*, blk_status_t *error*)
Helper function for drives to finish the request.

Parameters

struct request * rq the request to finish

blk_status_t error block status code

Description

Completely finish **rq**.

bool **__blk_end_request**(struct request * *rq*, blk_status_t *error*, unsigned int *nr_bytes*)
Helper function for drivers to complete the request.

Parameters

struct request * rq the request being processed

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete

Description

Must be called with queue lock held unlike *blk_end_request()*.

Return

`false` - we are done with this request `true` - still buffers pending for this request

void **__blk_end_request_all**(struct request * *rq*, blk_status_t *error*)
Helper function for drives to finish the request.

Parameters

struct request * rq the request to finish

blk_status_t error block status code

Description

Completely finish **rq**. Must be called with queue lock held.

bool **__blk_end_request_cur**(struct request * *rq*, blk_status_t *error*)
Helper function to finish the current request chunk.

Parameters

struct request * rq the request to finish the current chunk for

blk_status_t error block status code

Description

Complete the current consecutively mapped chunk from **rq**. Must be called with queue lock held.

Return

`false` - we are done with this request `true` - still buffers pending for this request

void **rq_flush_dcache_pages**(struct request * *rq*)
Helper function to flush all pages in a request

Parameters

struct request * rq the request to be flushed

Description

Flush all pages in **rq**.

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_rq_unprep_clone**(struct request * *rq*)
Helper function to free all bios in a cloned request

Parameters

struct request * rq the clone request to be cleaned up

Description

Free all bios in **rq** for a cloned request.

int **blk_rq_prep_clone**(struct request * *rq*, struct request * *rq_src*, struct bio_set * *bs*, gfp_t *gfp_mask*, int (*bio_ctr) (struct bio *, struct bio *, void *, void * *data*)
Helper function to setup clone request

Parameters

struct request * rq the request to be setup

struct request * rq_src original request to be cloned

struct bio_set * bs bio_set that bios for clone are allocated from

gfp_t gfp_mask memory allocation mask for bio

int (*)(struct bio *, struct bio *, void *) bio_ctr setup function to be called for each clone bio. Returns `0` for success, non `0` for failure.

void * data private data to be passed to **bio_ctr**

Description

Clones bios in **rq_src** to **rq**, and copies attributes of **rq_src** to **rq**. The actual data parts of **rq_src** (e.g. ->cmd, ->sense) are not copied, and copying such parts is the caller's responsibility. Also, pages which the original bios are pointing to are not copied and the cloned bios just point same pages. So cloned bios must be completed before original bios, which means the caller must complete **rq** before **rq_src**.

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

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_pm_runtime_init**(struct request_queue * *q*, struct device * *dev*)
Block layer runtime PM initialization routine

Parameters

struct request_queue * q the queue of the device

struct device * dev the device the queue belongs to

Description

Initialize runtime-PM-related fields for **q** and start auto suspend for **dev**. Drivers that want to take advantage of request-based runtime PM should call this function after **dev** has been initialized, and its request queue **q** has been allocated, and runtime PM for it can not happen yet(either due to disabled/forbidden or its usage_count > 0). In most cases, driver should call this function before any I/O has taken place.

This function takes care of setting up using auto suspend for the device, the autosuspend delay is set to -1 to make runtime suspend impossible until an updated value is either set by user or by driver. Drivers do not need to touch other autosuspend settings.

The block layer runtime PM is request based, so only works for drivers that use request as their IO unit instead of those directly use bio's.

int **blk_pre_runtime_suspend**(struct request_queue * *q*)
Pre runtime suspend check

Parameters

struct request_queue * q the queue of the device

Description

This function will check if runtime suspend is allowed for the device by examining if there are any requests pending in the queue. If there are requests pending, the device can not be runtime suspended; otherwise, the queue's status will be updated to SUSPENDING and the driver can proceed to suspend the device.

For the not allowed case, we mark last busy for the device so that runtime PM core will try to autosuspend it some time later.

This function should be called near the start of the device's runtime_suspend callback.

Return

0 - OK to runtime suspend the device -EBUSY - Device should not be runtime suspended

void **blk_post_runtime_suspend**(struct request_queue * *q*, int *err*)
Post runtime suspend processing

Parameters

struct request_queue * q the queue of the device

int err return value of the device's runtime_suspend function

Description

Update the queue's runtime status according to the return value of the device's runtime suspend function and mark last busy for the device so that PM core will try to auto suspend the device at a later time.

This function should be called near the end of the device's runtime_suspend callback.

void **blk_pre_runtime_resume**(struct request_queue * *q*)
Pre runtime resume processing

Parameters

struct request_queue * q the queue of the device

Description

Update the queue's runtime status to RESUMING in preparation for the runtime resume of the device.

This function should be called near the start of the device's runtime_resume callback.

void **blk_post_runtime_resume**(struct request_queue * *q*, int *err*)
Post runtime resume processing

Parameters

struct request_queue * q the queue of the device

int err return value of the device's runtime_resume function

Description

Update the queue's runtime status according to the return value of the device's runtime_resume function. If it is successfully resumed, process the requests that are queued into the device's queue when it is resuming and then mark last busy and initiate autosuspend for it.

This function should be called near the end of the device's runtime_resume callback.

void **blk_set_runtime_active**(struct request_queue * *q*)
Force runtime status of the queue to be active

Parameters

struct request_queue * q the queue of the device

Description

If the device is left runtime suspended during system suspend the resume hook typically resumes the device and corrects runtime status accordingly. However, that does not affect the queue runtime PM status which is still "suspended". This prevents processing requests from the queue.

This function can be used in driver's resume hook to correct queue runtime PM status and re-enable peeking requests from the queue. It should be called before first request is added to the queue.

void **__blk_drain_queue**(struct request_queue * *q*, bool *drain_all*)
drain requests from request_queue

Parameters

struct request_queue * q queue to drain

bool drain_all whether to drain all requests or only the ones w/ ELVPRIV

Description

Drain requests from **q**. If **drain_all** is set, all requests are drained. If not, only ELVPRIV requests are drained. The caller is responsible for ensuring that no new requests which need to be drained are queued.

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_PREEMPT

struct request * **__get_request**(struct request_list * *rl*, unsigned int *op*, struct bio * *bio*, blk_mq_req_flags_t *flags*, gfp_t *gfp_mask*)
get a free request

Parameters

struct request_list * rl request list to allocate from

unsigned int op operation and flags

struct bio * bio bio to allocate request for (can be NULL)

blk_mq_req_flags_t flags BLQ_MQ_REQ_* flags

gfp_t gfp_mask allocator flags

Description

Get a free request from **q**. This function may fail under memory pressure or if **q** is dead.

Must be called with **q**->queue_lock held and, Returns ERR_PTR on failure, with **q**->queue_lock held. Returns request pointer on success, with **q**->queue_lock *not held*.

struct request * **get_request**(struct request_queue * *q*, unsigned int *op*, struct bio * *bio*, blk_mq_req_flags_t *flags*, gfp_t *gfp*)
get a free request

Parameters

struct request_queue * q request_queue to allocate request from

unsigned int op operation and flags

struct bio * bio bio to allocate request for (can be NULL)

blk_mq_req_flags_t flags BLK_MQ_REQ_* flags.

gfp_t gfp allocator flags

Description

Get a free request from **q**. If BLK_MQ_REQ_NOWAIT is set in **flags**, this function keeps retrying under memory pressure and fails iff **q** is dead.

Must be called with **q**->queue_lock held and, Returns ERR_PTR on failure, with **q**->queue_lock held. Returns request pointer on success, with **q**->queue_lock *not held*.

bool **blk_attempt_plug_merge**(struct request_queue * *q*, struct bio * *bio*, unsigned int * *request_count*, struct request ** *same_queue_rq*)
try to merge with current's plugged list

Parameters

struct request_queue * q request_queue new bio is being queued at

struct bio * bio new bio being queued

unsigned int * request_count out parameter for number of traversed plugged requests

struct request ** same_queue_rq pointer to struct request that gets filled in when another request associated with **q** is found on the plug list (optional, may be NULL)

Description

Determine whether **bio** being queued on **q** can be merged with a request on current's plugged list. Returns true if merge was successful, otherwise false.

Plugging coalesces IOs from the same issuer for the same purpose without going through **q**->queue_lock. As such it's more of an issuing mechanism than scheduling, and the request, while may have elvpriv data, is not added on the elevator at this point. In addition, we don't have reliable access to the elevator outside queue lock. Only check basic merging parameters without querying the elevator.

Caller must ensure !blk_queue_nomerges(q) beforehand.

int **blk_cloned_rq_check_limits**(struct request_queue * *q*, struct request * *rq*)
Helper function to check a cloned request for new the queue limits

Parameters

struct request_queue * q the queue

struct request * rq the request being checked

Description

rq may have been made based on weaker limitations of upper-level queues in request stacking drivers, and it may violate the limitation of **q**. Since the block layer and the underlying device driver trust **rq** after it is inserted to **q**, it should be checked against **q** before the insertion using this generic function.

Request stacking drivers like request-based dm may change the queue limits when retrying requests on other queues. Those requests need to be checked against the new queue limits again during dispatch.

bool **blk_end_bidi_request**(struct request * *rq*, blk_status_t *error*, unsigned int *nr_bytes*, unsigned int *bidi_bytes*)
Complete a bidi request

Parameters

struct request * rq the request to complete

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete **rq**

unsigned int bidi_bytes number of bytes to complete **rq**->next_rq

Description

Ends I/O on a number of bytes attached to **rq** and **rq**->next_rq. Drivers that supports bidi can safely call this member for any type of request, bidi or uni. In the later case **bidi_bytes** is just ignored.

Return

`false` - we are done with this request `true` - still buffers pending for this request

bool **__blk_end_bidi_request**(struct request * *rq*, blk_status_t *error*, unsigned int *nr_bytes*, unsigned int *bidi_bytes*)
Complete a bidi request with queue lock held

Parameters

struct request * rq the request to complete

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete **rq**

unsigned int bidi_bytes number of bytes to complete **rq**->next_rq

Description

Identical to *blk_end_bidi_request()* except that queue lock is assumed to be locked on entry and remains so on return.

Return

`false` - we are done with this request `true` - still buffers pending for this request

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.

Note

The mapped bio may need to be bounced through blk_queue_bounce() before being submitted to the device, as pages mapped may be out of reach. It's the callers responsibility to make sure this happens. The original bio must be passed back in to *blk_rq_unmap_user()* for proper unmapping.

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.

void **__blk_release_queue**(struct work_struct * *work*)
release a request queue when it is no longer needed

Parameters

struct work_struct * work pointer to the release_work member of the request queue to be released

Description

blk_release_queue is the counterpart of *blk_init_queue()*. It should be called when a request queue is being released; typically when a block device is being de-registered. Its primary task it to free the queue itself.

Notes

The low level driver must have finished any outstanding requests first via *blk_cleanup_queue()*.

Although `blk_release_queue()` may be called with preemption disabled, *__blk_release_queue()* may sleep.

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_queue_prep_rq**(struct request_queue * *q*, prep_rq_fn * *pfn*)
set a prepare_request function for queue

Parameters

struct request_queue * q queue

prep_rq_fn * pfn prepare_request function

Description

It's possible for a queue to register a prepare_request callback which is invoked before the request is handed to the request_fn. The goal of the function is to prepare a request for I/O, it can be used to build a cdb from the request data for instance.

void **blk_queue_unprep_rq**(struct request_queue * *q*, unprep_rq_fn * *ufn*)
set an unprepare_request function for queue

Parameters

struct request_queue * q queue

unprep_rq_fn * ufn unprepare_request function

Description

It's possible for a queue to register an unprepare_request callback which is invoked before the request is finally completed. The goal of the function is to deallocate any data that was allocated in the prepare_request callback.

void **blk_set_default_limits**(struct queue_limits * *lim*)
reset limits to default values

Parameters

struct queue_limits * lim the queue_limits structure to reset

Description

Returns a queue_limit struct to its default state.

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_make_request**(struct request_queue * *q*, make_request_fn * *mfn*)
define an alternate make_request function for a device

Parameters

struct request_queue * q the request queue for the device to be affected

make_request_fn * mfn the alternate make_request function

Description

The normal way for `struct bios` to be passed to a device driver is for them to be collected into requests on a request queue, and then to allow the device driver to select requests off that queue when it is ready. This works well for many block devices. However some block devices (typically virtual devices such as md or lvm) do not benefit from the processing on the request queue, and are served best by having the requests passed directly to them. This can be achieved by providing a function to `blk_queue_make_request()`.

Caveat: The driver that does this *must* be able to deal appropriately with buffers in "highmemory". This can be accomplished by either calling `kmap_atomic()` to get a temporary kernel mapping, or by calling `blk_queue_bounce()` to create a buffer in normal memory.

void **blk_queue_bounce_limit**(struct request_queue * *q*, u64 *max_addr*)
set bounce buffer limit for queue

Parameters

struct request_queue * q the request queue for the device

u64 max_addr the maximum address the device can handle

Description

Different hardware can have different requirements as to what pages it can do I/O directly to. A low level driver can call blk_queue_bounce_limit to have lower memory pages allocated as bounce buffers for doing I/O to pages residing above **max_addr**.

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 chunk size must currently be a power-of-2 in sectors. Also 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_write_same_sectors**(struct request_queue * *q*, unsigned int *max_write_same_sectors*)
set max sectors for a single write same

Parameters

struct request_queue * q the request queue for the device

unsigned int max_write_same_sectors maximum number of sectors to write per command

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_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 short *size*)
set logical block size for the queue

Parameters

struct request_queue * q the request queue for the device

unsigned short 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_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.

void **blk_queue_stack_limits**(struct request_queue * *t*, struct request_queue * *b*)
inherit underlying queue limits for stacked drivers

Parameters

struct request_queue * t the stacking driver (top)

struct request_queue * b the underlying device (bottom)

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.

int **bdev_stack_limits**(struct queue_limits * *t*, struct block_device * *bdev*, sector_t *start*)
adjust queue limits for stacked drivers

Parameters

struct queue_limits * t the stacking driver limits (top device)

struct block_device * bdev the component block_device (bottom)

sector_t start first data sector within component device

Description

Merges queue limits for a top device and a block_device. Returns 0 if alignment didn't change. Returns -1 if adding the bottom device caused misalignment.

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_dma_pad**(struct request_queue * *q*, unsigned int *mask*)
set pad mask

Parameters

struct request_queue * q the request queue for the device

unsigned int mask pad mask

Description

Set 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_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.

int **blk_queue_dma_drain**(struct request_queue * *q*, dma_drain_needed_fn * *dma_drain_needed*, void * *buf*, unsigned int *size*)
Set up a drain buffer for excess dma.

Parameters

struct request_queue * q the request queue for the device

dma_drain_needed_fn * dma_drain_needed fn which returns non-zero if drain is necessary

void * buf physically contiguous buffer

unsigned int size size of the buffer in bytes

Description

Some devices have excess DMA problems and can't simply discard (or zero fill) the unwanted piece of the transfer. They have to have a real area of memory to transfer it into. The use case for this is ATAPI devices in DMA mode. If the packet command causes a transfer bigger than the transfer size some HBAs will lock up if there aren't DMA elements to contain the excess transfer. What this API does is adjust the queue so that the buf is always appended silently to the scatterlist.

Note

This routine adjusts max_hw_segments to make room for appending the drain buffer. If you call `blk_queue_max_segments()` after calling this routine, you must set the limit to one fewer than your device can support otherwise there won't be room for the drain 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_execute_rq_nowait**(struct request_queue * *q*, struct gendisk * *bd_disk*, struct request * *rq*, int *at_head*, rq_end_io_fn * *done*)
insert a request into queue for execution

Parameters

struct request_queue * q queue to insert the request in

struct gendisk * bd_disk matching gendisk

struct request * rq request to insert

int at_head insert request at head or tail of queue

rq_end_io_fn * done I/O completion handler

Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution. Don't wait for completion.

Note

This function will invoke **done** directly if the queue is dead.

void **blk_execute_rq**(struct request_queue * *q*, struct gendisk * *bd_disk*, struct request * *rq*, int *at_head*)
insert a request into queue for execution

Parameters

struct request_queue * q queue to insert the request in

struct gendisk * bd_disk matching gendisk

struct request * rq request to insert

int at_head insert request at head or tail of queue

Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution and wait for completion.

int **blkdev_issue_flush**(struct block_device * *bdev*, gfp_t *gfp_mask*, sector_t * *error_sector*)
queue a flush

Parameters

struct block_device * bdev blockdev to issue flush for

gfp_t gfp_mask memory allocation flags (for bio_alloc)

sector_t * error_sector error sector

Description

Issue a flush for the block device in question. Caller can supply room for storing the error offset in case of a flush error, if they wish to.

int **blkdev_issue_discard**(struct block_device * *bdev*, sector_t *sector*, sector_t *nr_sects*, gfp_t *gfp_mask*, unsigned long *flags*)
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)

unsigned long flags BLKDEV_DISCARD_* flags to control behaviour

Description

Issue a discard request for the sectors in question.

int **blkdev_issue_write_same**(struct block_device * *bdev*, sector_t *sector*, sector_t *nr_sects*, gfp_t *gfp_mask*, struct page * *page*)
queue a write same operation

Parameters

struct block_device * bdev target blockdev

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 page * page page containing data

Description

Issue a write same 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`.

struct request * **blk_queue_find_tag**(struct request_queue * *q*, int *tag*)
find a request by its tag and queue

Parameters

struct request_queue * q The request queue for the device

int tag The tag of the request

Notes

Should be used when a device returns a tag and you want to match it with a request.

no locks need be held.

void **blk_free_tags**(struct blk_queue_tag * *bqt*)
release a given set of tag maintenance info

Parameters

struct blk_queue_tag * bqt the tag map to free

Description

Drop the reference count on **bqt** and frees it when the last reference is dropped.

void **blk_queue_free_tags**(struct request_queue * *q*)
release tag maintenance info

Parameters

struct request_queue * q the request queue for the device

Notes

This is used to disable tagged queuing to a device, yet leave queue in function.

struct blk_queue_tag * **blk_init_tags**(int *depth*, int *alloc_policy*)
initialize the tag info for an external tag map

Parameters

int depth the maximum queue depth supported

int alloc_policy tag allocation policy

int **blk_queue_init_tags**(struct request_queue * *q*, int *depth*, struct blk_queue_tag * *tags*, int *alloc_policy*)
initialize the queue tag info

Parameters

struct request_queue * q the request queue for the device

int depth the maximum queue depth supported

struct blk_queue_tag * tags the tag to use

int alloc_policy tag allocation policy

Description

Queue lock must be held here if the function is called to resize an existing map.

int **blk_queue_resize_tags**(struct request_queue * *q*, int *new_depth*)
change the queueing depth

Parameters

struct request_queue * q the request queue for the device

int new_depth the new max command queueing depth

Notes

Must be called with the queue lock held.

int **blk_queue_start_tag**(struct request_queue * *q*, struct request * *rq*)
find a free tag and assign it

Parameters

struct request_queue * q the request queue for the device

struct request * rq the block request that needs tagging

Description

This can either be used as a stand-alone helper, or possibly be assigned as the queue `prep_rq_fn` (in which case `struct request` automagically gets a tag assigned). Note that this function assumes that any type of request can be queued! if this is not true for your device, you must check the request type before calling this function. The request will also be removed from the request queue, so it's the drivers responsibility to readd it if it should need to be restarted for some reason.

void **__blk_queue_free_tags**(struct request_queue * *q*)
release tag maintenance info

Parameters

struct request_queue * q the request queue for the device

Notes

blk_cleanup_queue() will take care of calling this function, if tagging has been used. So there's no need to call this directly.

void **blk_queue_end_tag**(struct request_queue * *q*, struct request * *rq*)
end tag operations for a request

Parameters

struct request_queue * q the request queue for the device

struct request * rq the request that has completed

Description

Typically called when `end_that_request_first()` returns `0`, meaning all transfers have been done for a request. It's important to call this function before `end_that_request_last()`, as that will put the request back on the free list thus corrupting the internal tag list.

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.txt.

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*, int *error*, unsigned int *nr_bytes*, u32 *what*, union kernfs_node_id * *cgid*)
Add a trace for a request oriented action

Parameters

struct request * rq the source request

int error return status to log

unsigned int nr_bytes number of completed bytes

u32 what the action

union kernfs_node_id * 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 request_queue * *q*, 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 request_queue * q queue the io is for

struct bio * bio the source bio

dev_t dev target device

sector_t from source sector

Description

Device mapper or raid target sometimes need to split a bio because it spans a stripe (or similar). Add a trace for that action.

void **blk_add_trace_rq_remap**(void * *ignore*, struct request_queue * *q*, 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_queue * q queue the io is for

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.

int **blk_mangle_minor**(int *minor*)
scatter minor numbers apart

Parameters

int minor minor number to mangle

Description

Scatter consecutively allocated **minor** number apart if MANGLE_DEVT is enabled. Mangling twice gives the original value.

Return

Mangled value.

Context

Don't care.

int **blk_alloc_devt**(struct hd_struct * *part*, dev_t * *devt*)
allocate a dev_t for a partition

Parameters

struct hd_struct * part partition to allocate dev_t for

dev_t * devt out parameter for resulting dev_t

Description

Allocate a dev_t for block device.

Return

0 on success, allocated dev_t is returned in ***devt**. -errno on failure.

Context

Might sleep.

void **blk_free_devt**(dev_t *devt*)
free a dev_t

Parameters

dev_t devt dev_t to free

Description

Free **devt** which was allocated using *blk_alloc_devt()*.

Context

Might sleep.

void **__device_add_disk**(struct device * *parent*, struct gendisk * *disk*, bool *register_queue*)
add disk information to kernel list

Parameters

struct device * parent parent device for the disk

struct gendisk * disk per-device partitioning information

bool register_queue register the queue if set to true

Description

This function registers the partitioning information in **disk** with the kernel.

FIXME: error handling

void **disk_replace_part_tbl**(struct gendisk * *disk*, struct disk_part_tbl * *new_ptbl*)
replace disk->part_tbl in RCU-safe way

Parameters

struct gendisk * disk disk to replace part_tbl for

struct disk_part_tbl * new_ptbl new part_tbl to install

Description

Replace disk->part_tbl with **new_ptbl** in RCU-safe way. The original ptbl is freed using RCU callback.

LOCKING: Matching bd_mutex locked or the caller is the only user of **disk**.

int **disk_expand_part_tbl**(struct gendisk * *disk*, int *partno*)
expand disk->part_tbl

Parameters

struct gendisk * disk disk to expand part_tbl for

int partno expand such that this partno can fit in

Description

Expand disk->part_tbl such that **partno** can fit in. disk->part_tbl uses RCU to allow unlocked dereferencing for stats and other stuff.

LOCKING: Matching bd_mutex locked or the caller is the only user of **disk**. Might sleep.

Return

0 on success, -errno on failure.

void **disk_block_events**(struct gendisk * *disk*)
block and flush disk event checking

Parameters

struct gendisk * disk disk to block events for

Description

On return from this function, it is guaranteed that event checking isn't in progress and won't happen until unblocked by *disk_unblock_events()*. Events blocking is counted and the actual unblocking happens after the matching number of unblocks are done.

Note that this intentionally does not block event checking from *disk_clear_events()*.

Context

Might sleep.

void **disk_unblock_events**(struct gendisk * *disk*)
unblock disk event checking

Parameters

struct gendisk * disk disk to unblock events for

Description

Undo *disk_block_events()*. When the block count reaches zero, it starts events polling if configured.

Context

Don't care. Safe to call from irq context.

void **disk_flush_events**(struct gendisk * *disk*, unsigned int *mask*)
schedule immediate event checking and flushing

Parameters

struct gendisk * disk disk to check and flush events for

unsigned int mask events to flush

Description

Schedule immediate event checking on **disk** if not blocked. Events in **mask** are scheduled to be cleared from the driver. Note that this doesn't clear the events from **disk**->ev.

Context

If **mask** is non-zero must be called with bdev->bd_mutex held.

unsigned int **disk_clear_events**(struct gendisk * *disk*, unsigned int *mask*)
synchronously check, clear and return pending events

Parameters

struct gendisk * disk disk to fetch and clear events from

unsigned int mask mask of events to be fetched and cleared

Description

Disk events are synchronously checked and pending events in **mask** are cleared and returned. This ignores the block count.

Context

Might sleep.

struct hd_struct * **disk_get_part**(struct gendisk * *disk*, int *partno*)
get partition

Parameters

struct gendisk * disk disk to look partition from

int partno partition number

Description

Look for partition **partno** from **disk**. If found, increment reference count and return it.

Context

Don't care.

Return

Pointer to the found partition on success, NULL if not found.

void **disk_part_iter_init**(struct disk_part_iter * *piter*, struct gendisk * *disk*, unsigned int *flags*)
initialize partition iterator

Parameters

struct disk_part_iter * piter iterator to initialize

struct gendisk * disk disk to iterate over

unsigned int flags DISK_PITER_* flags

Description

Initialize **piter** so that it iterates over partitions of **disk**.

Context

Don't care.

struct hd_struct * **disk_part_iter_next**(struct disk_part_iter * *piter*)
proceed iterator to the next partition and return it

Parameters

struct disk_part_iter * piter iterator of interest

Description

Proceed **piter** to the next partition and return it.

Context

Don't care.

void **disk_part_iter_exit**(struct disk_part_iter * *piter*)
finish up partition iteration

Parameters

struct disk_part_iter * piter iter of interest

Description

Called when iteration is over. Cleans up **piter**.

Context

Don't care.

struct hd_struct * **disk_map_sector_rcu**(struct gendisk * *disk*, sector_t *sector*)
map sector to partition

Parameters

struct gendisk * disk gendisk of interest

sector_t sector sector to map

Description

Find out which partition **sector** maps to on **disk**. This is primarily used for stats accounting.

Context

RCU read locked. The returned partition pointer is valid only while preemption is disabled.

Return

Found partition on success, part0 is returned if no partition matches

int **register_blkdev**(unsigned int *major*, const char * *name*)
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

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.

struct gendisk * **get_gendisk**(dev_t *devt*, int * *partno*)
get partitioning information for a given device

Parameters

dev_t devt device to get partitioning information for

int * partno returned partition index

Description

This function gets the structure containing partitioning information for the given device **devt**.

struct block_device * **bdget_disk**(struct gendisk * *disk*, int *partno*)
do `bdget()` by gendisk and partition number

Parameters

struct gendisk * disk gendisk of interest

int partno partition number

Description

Find partition **partno** from **disk**, do `bdget()` on it.

Context

Don't care.

Return

Resulting block_device on success, NULL on failure.

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()*.

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*)
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*)
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

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.

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_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.

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 **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.

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

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.

devm_clk_get should not be called from within interrupt context.

The clock will automatically be 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_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_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.

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.

clk_bulk_put 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.

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

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**(struct clk * *clk*, struct clk * *parent*)
check if a clock is a possible parent for another

Parameters

struct clk * clk clock source

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.

Synchronization Primitives

Read-Copy Update (RCU)

RCU_NONIDLE(*a*)
Indicate idle-loop code that needs RCU readers

Parameters

a Code that RCU needs to pay attention to.

Description

RCU, RCU-bh, and RCU-sched read-side critical sections are forbidden in the inner idle loop, that is, between the *rcu_idle_enter()* and the *rcu_idle_exit()* – RCU will happily ignore any such read-side critical sections. However, things like powertop need tracepoints in the inner idle loop.

This macro provides the way out: RCU_NONIDLE(`do_something_with_RCU()`) will tell RCU that it needs to pay attention, invoke its argument (in this example, calling the `do_something_with_RCU()` function), and then tell RCU to go back to ignoring this CPU. It is permissible to nest *RCU_NONIDLE()* wrappers, but not indefinitely (but the limit is on the order of a million or so, even on 32-bit systems). It is not legal to block within *RCU_NONIDLE()*, nor is it permissible to transfer control either into or out of *RCU_NONIDLE()*'s statement.

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 PREEMPT kernels.

RCU_LOCKDEP_WARN(*c*, *s*)
emit lockdep splat if specified condition is met

Parameters

c condition to check

s informative message

RCU_INITIALIZER(*v*)
statically initialize an RCU-protected global variable

Parameters

v The value to statically initialize with.

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_swap_protected(*rcu_ptr*, *ptr*, *c*)
swap an RCU and a regular pointer

Parameters

rcu_ptr RCU pointer

ptr regular pointer

c the conditions under which the dereference will take place

Description

Perform swap(**rcu_ptr**, **ptr**) where **rcu_ptr** is an RCU-annotated pointer and **c** is the argument that is passed to the *rcu_dereference_protected()* call used to read that 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.

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.

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(*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()*.

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()*.

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(*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(*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(*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(*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.

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 !PREEMPT kernel. But if you want the full story, read on!

In non-preemptible RCU implementations (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_PREEMPT 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 most situations, *rcu_read_unlock()* is immune from deadlock. However, in kernels built with CONFIG_RCU_BOOST, *rcu_read_unlock()* is responsible for deboosting, which it does via `rt_mutex_unlock()`. Unfortunately, this function acquires the scheduler’s runqueue and priority-inheritance spinlocks. This means that deadlock could result if the caller of *rcu_read_unlock()* already holds one of these locks or any lock that is ever acquired while holding them.

That said, RCU readers are never priority boosted unless they were preempted. Therefore, one way to avoid deadlock is to make sure that preemption never happens within any RCU read-side critical section whose outermost *rcu_read_unlock()* is called with one of `rt_mutex_unlock()`'s locks held. Such preemption can be avoided in a number of ways, for example, by invoking `preempt_disable()` before critical section's outermost *rcu_read_lock()*.

Given that the set of locks acquired by `rt_mutex_unlock()` might change at any time, a somewhat more future-proofed approach is to make sure that that preemption never happens within any RCU read-side critical section whose outermost *rcu_read_unlock()* is called with irqs disabled. This approach relies on the fact that `rt_mutex_unlock()` currently only acquires irq-disabled locks.

The second of these two approaches is best in most situations, however, the first approach can also be useful, at least to those developers willing to keep abreast of the set of locks acquired by `rt_mutex_unlock()`.

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 of *rcu_read_lock()*, but to be used when updates are being done using `call_rcu_bh()` or `synchronize_rcu_bh()`. Since both `call_rcu_bh()` and `synchronize_rcu_bh()` consider completion of a softirq handler to be a quiescent state, a process in RCU read-side critical section must be protected by disabling softirqs. Read-side critical sections in interrupt context can use just *rcu_read_lock()*, though this should at least be commented to avoid confusing people reading the code.

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_lock_sched**(void)
mark the beginning of a RCU-sched critical section

Parameters

void no arguments

Description

This is equivalent of *rcu_read_lock()*, but to be used when updates are being done using `call_rcu_sched()` or `synchronize_rcu_sched()`. Read-side critical sections can also be introduced by anything that disables preemption, including `local_irq_disable()` and friends.

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.

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(*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(*ptr*, *rcu_head*)
kfree an object after a grace period.

Parameters

ptr pointer to kfree

rcu_head 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 `__kfree_rcu()`. 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.

Note that the allowable offset might decrease in the future, for example, to allow something like `kmem_cache_free_rcu()`.

The BUILD_BUG_ON check must not involve any function calls, hence the checks are done in macros here.

synchronize_rcu_mult(*...*)
Wait concurrently for multiple grace periods

Parameters

... List of `call_rcu()` functions for the flavors to wait on.

Description

This macro waits concurrently for multiple flavors of RCU grace periods. For example, synchronize_rcu_mult(call_rcu, call_rcu_bh) would wait on concurrent RCU and RCU-bh grace periods. Waiting on a give SRCU domain requires you to write a wrapper function for that SRCU domain's *call_srcu()* function, supplying the corresponding srcu_struct.

If Tiny RCU, tell `_wait_rcu_gp()` not to bother waiting for RCU or RCU-bh, given that anywhere *synchronize_rcu_mult()* can be called is automatically a grace period.

void **synchronize_rcu_bh_expedited**(void)
Brute-force RCU-bh grace period

Parameters

void no arguments

Description

Wait for an RCU-bh grace period to elapse, but use a "big hammer" approach to force the grace period to end quickly. This consumes significant time on all CPUs and is 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_bh_expedited()* in a loop, please restructure your code to batch your updates, and then use a single `synchronize_rcu_bh()` instead.

Note that it is illegal to call this function while holding any lock that is acquired by a CPU-hotplug notifier. And yes, it is also illegal to call this function from a CPU-hotplug notifier. Failing to observe these restriction will result in deadlock.

void **rcu_idle_enter**(void)
inform RCU that current CPU is entering idle

Parameters

void no arguments

Description

Enter idle mode, in other words, -leave- the mode in which RCU read-side critical sections can occur. (Though RCU read-side critical sections can occur in irq handlers in idle, a possibility handled by `irq_enter()` and `irq_exit()`.)

If you add or remove a call to *rcu_idle_enter()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_user_enter**(void)
inform RCU that we are resuming userspace.

Parameters

void no arguments

Description

Enter RCU idle mode right before resuming userspace. No use of RCU is permitted between this call and *rcu_user_exit()*. This way the CPU doesn't need to maintain the tick for RCU maintenance purposes when the CPU runs in userspace.

If you add or remove a call to *rcu_user_enter()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_nmi_exit**(void)
inform RCU of exit from NMI context

Parameters

void no arguments

Description

If we are returning from the outermost NMI handler that interrupted an RCU-idle period, update rdtp->dynticks and rdtp->dynticks_nmi_nesting to let the RCU grace-period handling know that the CPU is back to being RCU-idle.

If you add or remove a call to *rcu_nmi_exit()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_irq_exit**(void)
inform RCU that current CPU is exiting irq towards idle

Parameters

void no arguments

Description

Exit from an interrupt handler, which might possibly result in entering idle mode, in other words, leaving the mode in which read-side critical sections can occur. The caller must have disabled interrupts.

This code assumes that the idle loop never does anything that might result in unbalanced calls to `irq_enter()` and `irq_exit()`. If your architecture's idle loop violates this assumption, RCU will give you what you deserve, good and hard. But very infrequently and irreproducibly.

Use things like work queues to work around this limitation.

You have been warned.

If you add or remove a call to *rcu_irq_exit()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_idle_exit**(void)
inform RCU that current CPU is leaving idle

Parameters

void no arguments

Description

Exit idle mode, in other words, -enter- the mode in which RCU read-side critical sections can occur.

If you add or remove a call to *rcu_idle_exit()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_user_exit**(void)
inform RCU that we are exiting userspace.

Parameters

void no arguments

Description

Exit RCU idle mode while entering the kernel because it can run a RCU read side critical section anytime.

If you add or remove a call to *rcu_user_exit()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_nmi_enter**(void)
inform RCU of entry to NMI context

Parameters

void no arguments

Description

If the CPU was idle from RCU's viewpoint, update rdtp->dynticks and rdtp->dynticks_nmi_nesting to let the RCU grace-period handling know that the CPU is active. This implementation permits nested NMIs, as long as the nesting level does not overflow an int. (You will probably run out of stack space first.)

If you add or remove a call to *rcu_nmi_enter()*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

void **rcu_irq_enter**(void)
inform RCU that current CPU is entering irq away from idle

Parameters

void no arguments

Description

Enter an interrupt handler, which might possibly result in exiting idle mode, in other words, entering the mode in which read-side critical sections can occur. The caller must have disabled interrupts.

Note that the Linux kernel is fully capable of entering an interrupt handler that it never exits, for example when doing upcalls to user mode! This code assumes that the idle loop never does upcalls to user mode. If your architecture's idle loop does do upcalls to user mode (or does anything else that results in unbalanced calls to the `irq_enter()` and `irq_exit()` functions), RCU will give you what you deserve, good and hard. But very infrequently and irreproducibly.

Use things like work queues to work around this limitation.

You have been warned.

If you add or remove a call to *`rcu_irq_enter()`*, be sure to test with CONFIG_RCU_EQS_DEBUG=y.

bool notrace **`rcu_is_watching`**(void)
see if RCU thinks that the current CPU is idle

Parameters

`void` no arguments

Description

Return true if RCU is watching the running CPU, which means that this CPU can safely enter RCU read-side critical sections. In other words, if the current CPU is in its idle loop and is neither in an interrupt or NMI handler, return true.

int **`rcu_is_cpu_rrupt_from_idle`**(void)
see if idle or immediately interrupted from idle

Parameters

`void` no arguments

Description

If the current CPU is idle or running at a first-level (not nested) interrupt from idle, return true. The caller must have at least disabled preemption.

void **`rcu_cpu_stall_reset`**(void)
prevent further stall warnings in current grace period

Parameters

`void` no arguments

Description

Set the stall-warning timeout way off into the future, thus preventing any RCU CPU stall-warning messages from appearing in the current set of RCU grace periods.

The caller must disable hard irqs.

void **`call_rcu_sched`**(struct rcu_head * *head*, rcu_callback_t *func*)
Queue an RCU for invocation after sched grace period.

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 currently executing RCU read-side critical sections have completed. `call_rcu_sched()` assumes that the read-side critical sections end on enabling of preemption or on voluntary preemption. RCU read-side critical sections are delimited by:

- *rcu_read_lock_sched()* and `rcu_read_unlock_sched()`, OR
- anything that disables preemption.

  These may be nested.

See the description of `call_rcu()` for more detailed information on memory ordering guarantees.

void **call_rcu_bh**(struct rcu_head * *head*, rcu_callback_t *func*)
Queue an RCU for invocation after a quicker grace period.

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 currently executing RCU read-side critical sections have completed. `call_rcu_bh()` assumes that the read-side critical sections end on completion of a softirq handler. This means that read-side critical sections in process context must not be interrupted by softirqs. This interface is to be used when most of the read-side critical sections are in softirq context. RCU read-side critical sections are delimited by:

- *rcu_read_lock()* and *rcu_read_unlock()*, if in interrupt context, OR
- *rcu_read_lock_bh()* and `rcu_read_unlock_bh()`, if in process context.

These may be nested.

See the description of `call_rcu()` for more detailed information on memory ordering guarantees.

void **synchronize_sched**(void)
wait until an rcu-sched grace period has elapsed.

Parameters

void no arguments

Description

Control will return to the caller some time after a full rcu-sched grace period has elapsed, in other words after all currently executing rcu-sched read-side critical sections have completed. These read-side critical sections are delimited by *rcu_read_lock_sched()* and `rcu_read_unlock_sched()`, and may be nested. Note that `preempt_disable()`, `local_irq_disable()`, and so on may be used in place of *rcu_read_lock_sched()*.

This means that all preempt_disable code sequences, including NMI and non-threaded hardware-interrupt handlers, in progress on entry will have completed before this primitive returns. However, this does not guarantee that softirq handlers will have completed, since in some kernels, these handlers can run in process context, and can block.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when `synchronize_sched()` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU-sched read-side critical section whose beginning preceded the call to `synchronize_sched()`. In addition, each CPU having an RCU read-side critical section that extends beyond the return from `synchronize_sched()` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_sched()` 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_sched()`, 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_sched()` – even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

void **synchronize_rcu_bh**(void)
wait until an rcu_bh grace period has elapsed.

Parameters

void no arguments

Description

Control will return to the caller some time after a full rcu_bh grace period has elapsed, in other words after all currently executing rcu_bh read-side critical sections have completed. RCU read-side critical sections are delimited by *rcu_read_lock_bh()* and rcu_read_unlock_bh(), and may be nested.

See the description of synchronize_sched() for more detailed information on memory ordering guarantees.

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() to determine whether or not a full grace period has elapsed in the meantime.

void **cond_synchronize_rcu**(unsigned long *oldstate*)
Conditionally wait for an RCU grace period

Parameters

unsigned long oldstate return value from earlier call to get_state_synchronize_rcu()

Description

If a full RCU grace period has elapsed since the earlier call to get_state_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 one additional grace period should be just fine.

unsigned long **get_state_synchronize_sched**(void)
Snapshot current RCU-sched state

Parameters

void no arguments

Description

Returns a cookie that is used by a later call to cond_synchronize_sched() to determine whether or not a full grace period has elapsed in the meantime.

void **cond_synchronize_sched**(unsigned long *oldstate*)
Conditionally wait for an RCU-sched grace period

Parameters

unsigned long oldstate return value from earlier call to get_state_synchronize_sched()

Description

If a full RCU-sched grace period has elapsed since the earlier call to get_state_synchronize_sched(), just return. Otherwise, invoke synchronize_sched() 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 one additional grace period should be just fine.

void **rcu_barrier_bh**(void)
Wait until all in-flight call_rcu_bh() callbacks complete.

Parameters

void no arguments

void **rcu_barrier_sched**(void)
Wait for in-flight `call_rcu_sched()` callbacks.

Parameters

void no arguments

void **call_rcu**(struct rcu_head * *head*, rcu_callback_t *func*)
Queue an RCU callback for invocation after a grace period.

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.

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).

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.

See the description of `synchronize_sched()` for more detailed information on memory-ordering guarantees. However, please note that -only- the memory-ordering guarantees apply. For example, `synchronize_rcu()` is -not- guaranteed to wait on things like code protected by `preempt_disable()`, instead, `synchronize_rcu()` is -only- guaranteed to wait on RCU read-side critical sections, that is, sections of code protected by *rcu_read_lock()*.

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_sched_expedited**(void)
Brute-force RCU-sched grace period

Parameters

void no arguments

Description

Wait for an RCU-sched grace period to elapse, but use a "big hammer" approach to force the grace period to end quickly. This consumes significant time on all CPUs and is unfriendly to real-time workloads, so is thus not recommended for any sort of common-case code. In fact, if you are using *synchronize_sched_expedited()* in a loop, please restructure your code to batch your updates, and then use a single `synchronize_sched()` instead.

This implementation can be thought of as an application of sequence locking to expedited grace periods, but using the sequence counter to determine when someone else has already done the work instead of for retrying readers.

void **synchronize_rcu_expedited**(void)
Brute-force RCU grace period

Parameters

void no arguments

Description

Wait for an RCU-preempt 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-preempt critical section, and if so, it sets a flag that causes the outermost *rcu_read_unlock()* to report the quiescent state. 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 greate 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.

int **rcu_read_lock_sched_held**(void)
might we be in RCU-sched read-side critical section?

Parameters

void no arguments

Description

If CONFIG_DEBUG_LOCK_ALLOC is selected, returns 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 *rcu_idle_enter()* and *rcu_idle_exit()*) then `rcu_read_lock_held()` returns 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 SRCU read lock held if the current CPU is offline.

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()* 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.

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!), 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-strcuture 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.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when `synchronize_rcu_tasks()` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU-tasks read-side critical section whose beginning preceded the call to `synchronize_rcu_tasks()`. In addition, each CPU having an RCU-tasks read-side critical section that extends beyond the return from `synchronize_rcu_tasks()` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_rcu_tasks()` and before the beginning of that RCU-tasks 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_tasks()`, 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_tasks()` – even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

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 **`cleanup_srcu_struct`**(struct srcu_struct * *sp*)
deconstruct a sleep-RCU structure

Parameters

`struct srcu_struct * sp` 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 **`cleanup_srcu_struct_quiesced`**(struct srcu_struct * *sp*)
deconstruct a quiesced sleep-RCU structure

Parameters

`struct srcu_struct * sp` 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. Also, all grace-period processing must have completed.

"Completed" means that the last *`synchronize_srcu()`* and *`synchronize_srcu_expedited()`* calls must have returned before the call to *`cleanup_srcu_struct_quiesced()`*. It also means that the callback from the last *`call_srcu()`* must have been invoked before the call to *`cleanup_srcu_struct_quiesced()`*, but you can use *`srcu_barrier()`* to help with this last. Violating these rules will get you a `WARN_ON()` splat (with high probability, anyway), and will also cause the srcu_struct to be leaked.

int **`srcu_read_lock_held`**(const struct srcu_struct * *sp*)
might we be in SRCU read-side critical section?

Parameters

const struct srcu_struct * sp 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(*p*, *sp*, *c*)
fetch SRCU-protected pointer for later dereferencing

Parameters

p the pointer to fetch and protect for later dereferencing

sp 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(*p*, *sp*)
fetch SRCU-protected pointer for later dereferencing

Parameters

p the pointer to fetch and protect for later dereferencing

sp 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.

int **srcu_read_lock**(struct srcu_struct * *sp*)
register a new reader for an SRCU-protected structure.

Parameters

struct srcu_struct * sp 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.

void **srcu_read_unlock**(struct srcu_struct * *sp*, int *idx*)
unregister a old reader from an SRCU-protected structure.

Parameters

struct srcu_struct * sp 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 **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 * *sp*)
initialize a sleep-RCU structure

Parameters

struct srcu_struct * sp 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 * *sp*)
returns true if there are readers. and false otherwise

Parameters

struct srcu_struct * sp 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 **call_srcu**(struct srcu_struct * *sp*, struct rcu_head * *rhp*, rcu_callback_t *func*)
Queue a callback for invocation after an SRCU grace period

Parameters

struct srcu_struct * sp 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 * *sp*)
Brute-force SRCU grace period

Parameters

struct srcu_struct * sp 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 * *sp*)
    wait for prior SRCU read-side critical-section completion

Parameters

struct srcu_struct * sp 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-sched 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.

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.

void **srcu_barrier**(struct srcu_struct * *sp*)
    Wait until all in-flight *call_srcu()* callbacks complete.

Parameters

struct srcu_struct * sp srcu_struct on which to wait for in-flight callbacks.

unsigned long **srcu_batches_completed**(struct srcu_struct * *sp*)
    return batches completed.

Parameters

struct srcu_struct * sp 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_init_rcu**(struct hlist_bl_node * *n*)
deletes entry from hash list with re-initialization

Parameters

struct hlist_bl_node * n the element to delete from the hash list.

Note

`hlist_bl_unhashed()` on the node returns 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.

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_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()*.

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.

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(*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.

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.

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.

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 (*)(void) sync function to sync: `synchronize_rcu()`, `synchronize_sched()`, ...

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 (*)(void) sync function to sync: `synchronize_rcu()`, `synchronize_sched()`, ...

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 (*)(void) sync function to sync: `synchronize_rcu()`, `synchronize_sched()`, ...

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(*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(*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(*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_head within the struct.

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_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 (in which case updaters must use RCU-sched, as in `synchronize_sched()`, `call_rcu_sched()`, and friends). Another example is when items are added to the list, but never deleted.

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 (in which case updaters must use RCU-sched, as in `synchronize_sched()`, `call_rcu_sched()`, and friends). Another example is when items are added to the list, but never deleted.

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.

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.

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.

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 **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(*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_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(*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(*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(*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(*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 lockfree traversal if the writer side must know if the list entry is still hashed or already unhashed.

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()*.

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.

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()*.

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 for your 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/core-api/atomic_ops.rst around line 114 [2] Documentation/RCU/rculist_nulls.txt around line 146

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 for your 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 an RCU read-side critical section whose flavor matches that of the rcu_sync struture.

void **rcu_sync_init**(struct rcu_sync * *rsp*, enum rcu_sync_type *type*)
Initialize an rcu_sync structure

Parameters

struct rcu_sync * rsp Pointer to rcu_sync structure to be initialized

enum rcu_sync_type type Flavor of RCU with which to synchronize rcu_sync structure

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_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 slowpaths.

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_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 one of the `call_rcu()` functions by *rcu_sync_exit()*, so that it is invoked after a grace period following the that invocation of *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_exit**(struct rcu_sync * *rsp*)
Allow readers back onto fast patch 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

Generic Associative Array Implementation

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 whilst 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 whilst 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.

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 whilst 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.

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.

Semantics and Behavior of Atomic and Bitmask Operations

Author David S. Miller

This document is intended to serve as a guide to Linux port maintainers on how to implement atomic counter, bitops, and spinlock interfaces properly.

Atomic Type And Operations

The atomic_t type should be defined as a signed integer and the atomic_long_t type as a signed long integer. Also, they should be made opaque such that any kind of cast to a normal C integer type will fail. Something like the following should suffice:

```
typedef struct { int counter; } atomic_t;
typedef struct { long counter; } atomic_long_t;
```

Historically, counter has been declared volatile. This is now discouraged. See Documentation/process/volatile-considered-harmful.rst for the complete rationale.

local_t is very similar to atomic_t. If the counter is per CPU and only updated by one CPU, local_t is probably more appropriate. Please see *Documentation/core-api/local_ops.rst* for the semantics of local_t.

The first operations to implement for atomic_t's are the initializers and plain reads.

```
#define ATOMIC_INIT(i)          { (i) }
#define atomic_set(v, i)        ((v)->counter = (i))
```

The first macro is used in definitions, such as:

```
static atomic_t my_counter = ATOMIC_INIT(1);
```

The initializer is atomic in that the return values of the atomic operations are guaranteed to be correct reflecting the initialized value if the initializer is used before runtime. If the initializer is used at runtime, a proper implicit or explicit read memory barrier is needed before reading the value with atomic_read from another thread.

As with all of the `atomic_` interfaces, replace the leading `atomic_` with `atomic_long_` to operate on atomic_long_t.

The second interface can be used at runtime, as in:

```
struct foo { atomic_t counter; };
...

struct foo *k;

k = kmalloc(sizeof(*k), GFP_KERNEL);
if (!k)
        return -ENOMEM;
atomic_set(&k->counter, 0);
```

The setting is atomic in that the return values of the atomic operations by all threads are guaranteed to be correct reflecting either the value that has been set with this operation or set with another operation. A proper implicit or explicit memory barrier is needed before the value set with the operation is guaranteed to be readable with atomic_read from another thread.

Next, we have:

```
#define atomic_read(v)  ((v)->counter)
```

which simply reads the counter value currently visible to the calling thread. The read is atomic in that the return value is guaranteed to be one of the values initialized or modified with the interface operations if a proper implicit or explicit memory barrier is used after possible runtime initialization by any other thread and the value is modified only with the interface operations. atomic_read does not guarantee that the runtime initialization by any other thread is visible yet, so the user of the interface must take care of that with a proper implicit or explicit memory barrier.

Warning:

`atomic_read()` and `atomic_set()` DO NOT IMPLY BARRIERS!
Some architectures may choose to use the volatile keyword, barriers, or inline assembly to gua
tee some degree of immediacy for atomic_read() and atomic_set(). This is not uniformly guarant
*and may change in the future, so all users of atomic_t should treat atomic_read() and atomic_*
as simple C statements that may be reordered or optimized away entirely by the compiler or pro
sor, and explicitly invoke the appropriate compiler and/or memory barrier for each use case. Fa
to do so will result in code that may suddenly break when used with different architectures or
piler optimizations, or even changes in unrelated code which changes how the compiler optin
the section accessing atomic_t variables.

Properly aligned pointers, longs, ints, and chars (and unsigned equivalents) may be atomically loaded from and stored to in the same sense as described for atomic_read() and atomic_set(). The READ_ONCE() and WRITE_ONCE() macros should be used to prevent the compiler from using optimizations that might otherwise optimize accesses out of existence on the one hand, or that might create unsolicited accesses on the other.

For example consider the following code:

```
while (a > 0)
        do_something();
```

If the compiler can prove that do_something() does not store to the variable a, then the compiler is within its rights transforming this to the following:

```
if (a > 0)
        for (;;)
                do_something();
```

If you don't want the compiler to do this (and you probably don't), then you should use something like the following:

```
while (READ_ONCE(a) > 0)
        do_something();
```

Alternatively, you could place a barrier() call in the loop.

For another example, consider the following code:

```
tmp_a = a;
do_something_with(tmp_a);
do_something_else_with(tmp_a);
```

If the compiler can prove that do_something_with() does not store to the variable a, then the compiler is within its rights to manufacture an additional load as follows:

```
tmp_a = a;
do_something_with(tmp_a);
tmp_a = a;
do_something_else_with(tmp_a);
```

This could fatally confuse your code if it expected the same value to be passed to do_something_with() and do_something_else_with().

The compiler would be likely to manufacture this additional load if do_something_with() was an inline function that made very heavy use of registers: reloading from variable a could save a flush to the stack and later reload. To prevent the compiler from attacking your code in this manner, write the following:

```
tmp_a = READ_ONCE(a);
do_something_with(tmp_a);
do_something_else_with(tmp_a);
```

For a final example, consider the following code, assuming that the variable a is set at boot time before the second CPU is brought online and never changed later, so that memory barriers are not needed:

```
if (a)
        b = 9;
else
        b = 42;
```

The compiler is within its rights to manufacture an additional store by transforming the above code into the following:

```
b = 42;
if (a)
        b = 9;
```

This could come as a fatal surprise to other code running concurrently that expected b to never have the value 42 if a was zero. To prevent the compiler from doing this, write something like:

```
if (a)
        WRITE_ONCE(b, 9);
else
        WRITE_ONCE(b, 42);
```

Don't even -think- about doing this without proper use of memory barriers, locks, or atomic operations if variable a can change at runtime!

Warning: *READ_ONCE() OR WRITE_ONCE() DO NOT IMPLY A BARRIER!*

Now, we move onto the atomic operation interfaces typically implemented with the help of assembly code.

```
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

These four routines add and subtract integral values to/from the given atomic_t value. The first two routines pass explicit integers by which to make the adjustment, whereas the latter two use an implicit adjustment value of "1".

One very important aspect of these two routines is that they DO NOT require any explicit memory barriers. They need only perform the atomic_t counter update in an SMP safe manner.

Next, we have:

```
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

These routines add 1 and subtract 1, respectively, from the given atomic_t and return the new counter value after the operation is performed.

Unlike the above routines, it is required that these primitives include explicit memory barriers that are performed before and after the operation. It must be done such that all memory operations before and after the atomic operation calls are strongly ordered with respect to the atomic operation itself.

For example, it should behave as if a smp_mb() call existed both before and after the atomic operation.

If the atomic instructions used in an implementation provide explicit memory barrier semantics which satisfy the above requirements, that is fine as well.

Let's move on:

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
```

These behave just like atomic_{inc,dec}_return() except that an explicit counter adjustment is given instead of the implicit "1". This means that like atomic_{inc,dec}_return(), the memory barrier semantics are required.

Next:

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
```

These two routines increment and decrement by 1, respectively, the given atomic counter. They return a boolean indicating whether the resulting counter value was zero or not.

Again, these primitives provide explicit memory barrier semantics around the atomic operation:

```
int atomic_sub_and_test(int i, atomic_t *v);
```

This is identical to atomic_dec_and_test() except that an explicit decrement is given instead of the implicit "1". This primitive must provide explicit memory barrier semantics around the operation:

```
int atomic_add_negative(int i, atomic_t *v);
```

The given increment is added to the given atomic counter value. A boolean is return which indicates whether the resulting counter value is negative. This primitive must provide explicit memory barrier semantics around the operation.

Then:

```
int atomic_xchg(atomic_t *v, int new);
```

This performs an atomic exchange operation on the atomic variable v, setting the given new value. It returns the old value that the atomic variable v had just before the operation.

atomic_xchg must provide explicit memory barriers around the operation.

```
int atomic_cmpxchg(atomic_t *v, int old, int new);
```

This performs an atomic compare exchange operation on the atomic value v, with the given old and new values. Like all atomic_xxx operations, atomic_cmpxchg will only satisfy its atomicity semantics as long as all other accesses of *v are performed through atomic_xxx operations.

atomic_cmpxchg must provide explicit memory barriers around the operation, although if the comparison fails then no memory ordering guarantees are required.

The semantics for atomic_cmpxchg are the same as those defined for 'cas' below.

Finally:

```
int atomic_add_unless(atomic_t *v, int a, int u);
```

If the atomic value v is not equal to u, this function adds a to v, and returns non zero. If v is equal to u then it returns zero. This is done as an atomic operation.

atomic_add_unless must provide explicit memory barriers around the operation unless it fails (returns 0).

atomic_inc_not_zero, equivalent to atomic_add_unless(v, 1, 0)

If a caller requires memory barrier semantics around an atomic_t operation which does not return a value, a set of interfaces are defined which accomplish this:

```
void smp_mb__before_atomic(void);
void smp_mb__after_atomic(void);
```

Preceding a non-value-returning read-modify-write atomic operation with smp_mb__before_atomic() and following it with smp_mb__after_atomic() provides the same full ordering that is provided by value-returning read-modify-write atomic operations.

For example, smp_mb__before_atomic() can be used like so:

```
obj->dead = 1;
smp_mb__before_atomic();
atomic_dec(&obj->ref_count);
```

It makes sure that all memory operations preceding the atomic_dec() call are strongly ordered with respect to the atomic counter operation. In the above example, it guarantees that the assignment of "1" to obj->dead will be globally visible to other cpus before the atomic counter decrement.

Without the explicit smp_mb__before_atomic() call, the implementation could legally allow the atomic counter update visible to other cpus before the "obj->dead = 1;" assignment.

A missing memory barrier in the cases where they are required by the atomic_t implementation above can have disastrous results. Here is an example, which follows a pattern occurring frequently in the Linux kernel. It is the use of atomic counters to implement reference counting, and it works such that once the counter falls to zero it can be guaranteed that no other entity can be accessing the object:

```
static void obj_list_add(struct obj *obj, struct list_head *head)
{
        obj->active = 1;
        list_add(&obj->list, head);
}

static void obj_list_del(struct obj *obj)
```

```
{
        list_del(&obj->list);
        obj->active = 0;
}

static void obj_destroy(struct obj *obj)
{
        BUG_ON(obj->active);
        kfree(obj);
}

struct obj *obj_list_peek(struct list_head *head)
{
        if (!list_empty(head)) {
                struct obj *obj;

                obj = list_entry(head->next, struct obj, list);
                atomic_inc(&obj->refcnt);
                return obj;
        }
        return NULL;
}

void obj_poke(void)
{
        struct obj *obj;

        spin_lock(&global_list_lock);
        obj = obj_list_peek(&global_list);
        spin_unlock(&global_list_lock);

        if (obj) {
                obj->ops->poke(obj);
                if (atomic_dec_and_test(&obj->refcnt))
                        obj_destroy(obj);
        }
}

void obj_timeout(struct obj *obj)
{
        spin_lock(&global_list_lock);
        obj_list_del(obj);
        spin_unlock(&global_list_lock);

        if (atomic_dec_and_test(&obj->refcnt))
                obj_destroy(obj);
}
```

Note:

This is a simplification of the ARP queue management in the generic neighbour discover code of the networking. Olaf Kirch found a bug wrt. memory barriers in kfree_skb() that exposed the atomic_t memory barrier requirements quite clearly.

Given the above scheme, it must be the case that the obj->active update done by the obj list deletion be visible to other processors before the atomic counter decrement is performed.

Otherwise, the counter could fall to zero, yet obj->active would still be set, thus triggering the assertion in obj_destroy(). The error sequence looks like this:

```
cpu 0                           cpu 1
obj_poke()                      obj_timeout()
obj = obj_list_peek();
... gains ref to obj, refcnt=2
                                obj_list_del(obj);
                                obj->active = 0 ...
                                ... visibility delayed ...
                                atomic_dec_and_test()
                                ... refcnt drops to 1 ...
atomic_dec_and_test()
... refcount drops to 0 ...
obj_destroy()
BUG() triggers since obj->active
still seen as one
                                obj->active update visibility occurs
```

With the memory barrier semantics required of the atomic_t operations which return values, the above sequence of memory visibility can never happen. Specifically, in the above case the atomic_dec_and_test() counter decrement would not become globally visible until the obj->active update does.

As a historical note, 32-bit Sparc used to only allow usage of 24-bits of its atomic_t type. This was because it used 8 bits as a spinlock for SMP safety. Sparc32 lacked a "compare and swap" type instruction. However, 32-bit Sparc has since been moved over to a "hash table of spinlocks" scheme, that allows the full 32-bit counter to be realized. Essentially, an array of spinlocks are indexed into based upon the address of the atomic_t being operated on, and that lock protects the atomic operation. Parisc uses the same scheme.

Another note is that the atomic_t operations returning values are extremely slow on an old 386.

Atomic Bitmask

We will now cover the atomic bitmask operations. You will find that their SMP and memory barrier semantics are similar in shape and scope to the atomic_t ops above.

Native atomic bit operations are defined to operate on objects aligned to the size of an "unsigned long" C data type, and are least of that size. The endianness of the bits within each "unsigned long" are the native endianness of the cpu.

```
void set_bit(unsigned long nr, volatile unsigned long *addr);
void clear_bit(unsigned long nr, volatile unsigned long *addr);
void change_bit(unsigned long nr, volatile unsigned long *addr);
```

These routines set, clear, and change, respectively, the bit number indicated by "nr" on the bit mask pointed to by "ADDR".

They must execute atomically, yet there are no implicit memory barrier semantics required of these interfaces.

```
int test_and_set_bit(unsigned long nr, volatile unsigned long *addr);
int test_and_clear_bit(unsigned long nr, volatile unsigned long *addr);
int test_and_change_bit(unsigned long nr, volatile unsigned long *addr);
```

Like the above, except that these routines return a boolean which indicates whether the changed bit was set _BEFORE_ the atomic bit operation.

Warning: *It is incredibly important that the value be a boolean, ie. "0" or "1". Do not try to be fancy save a few instructions by declaring the above to return "long" and just returning something "old_val & mask" because that will not work.*

For one thing, this return value gets truncated to int in many code paths using these interfaces, so on 64-bit if the bit is set in the upper 32-bits then testers will never see that.

One great example of where this problem crops up are the thread_info flag operations. Routines such as test_and_set_ti_thread_flag() chop the return value into an int. There are other places where things like this occur as well.

These routines, like the atomic_t counter operations returning values, must provide explicit memory barrier semantics around their execution. All memory operations before the atomic bit operation call must be made visible globally before the atomic bit operation is made visible. Likewise, the atomic bit operation must be visible globally before any subsequent memory operation is made visible. For example:

```
obj->dead = 1;
if (test_and_set_bit(0, &obj->flags))
        /* ... */;
obj->killed = 1;
```

The implementation of test_and_set_bit() must guarantee that "obj->dead = 1;" is visible to cpus before the atomic memory operation done by test_and_set_bit() becomes visible. Likewise, the atomic memory operation done by test_and_set_bit() must become visible before "obj->killed = 1;" is visible.

Finally there is the basic operation:

```
int test_bit(unsigned long nr, __const__ volatile unsigned long *addr);
```

Which returns a boolean indicating if bit "nr" is set in the bitmask pointed to by "addr".

If explicit memory barriers are required around {set,clear}_bit() (which do not return a value, and thus does not need to provide memory barrier semantics), two interfaces are provided:

```
void smp_mb__before_atomic(void);
void smp_mb__after_atomic(void);
```

They are used as follows, and are akin to their atomic_t operation brothers:

```
/* All memory operations before this call will
 * be globally visible before the clear_bit().
 */
smp_mb__before_atomic();
clear_bit( ... );

/* The clear_bit() will be visible before all
 * subsequent memory operations.
 */
 smp_mb__after_atomic();
```

There are two special bitops with lock barrier semantics (acquire/release, same as spinlocks). These operate in the same way as their non-_lock/unlock postfixed variants, except that they are to provide acquire/release semantics, respectively. This means they can be used for bit_spin_trylock and bit_spin_unlock type operations without specifying any more barriers.

```
int test_and_set_bit_lock(unsigned long nr, unsigned long *addr);
void clear_bit_unlock(unsigned long nr, unsigned long *addr);
void __clear_bit_unlock(unsigned long nr, unsigned long *addr);
```

The __clear_bit_unlock version is non-atomic, however it still implements unlock barrier semantics. This can be useful if the lock itself is protecting the other bits in the word.

Finally, there are non-atomic versions of the bitmask operations provided. They are used in contexts where some other higher-level SMP locking scheme is being used to protect the bitmask, and thus less expensive non-atomic operations may be used in the implementation. They have names similar to the above bitmask operation interfaces, except that two underscores are prefixed to the interface name.

```
void __set_bit(unsigned long nr, volatile unsigned long *addr);
void __clear_bit(unsigned long nr, volatile unsigned long *addr);
void __change_bit(unsigned long nr, volatile unsigned long *addr);
int __test_and_set_bit(unsigned long nr, volatile unsigned long *addr);
int __test_and_clear_bit(unsigned long nr, volatile unsigned long *addr);
int __test_and_change_bit(unsigned long nr, volatile unsigned long *addr);
```

These non-atomic variants also do not require any special memory barrier semantics.

The routines xchg() and cmpxchg() must provide the same exact memory-barrier semantics as the atomic and bit operations returning values.

Note:

If someone wants to use xchg(), cmpxchg() and their variants, linux/atomic.h should be included rather than asm/cmpxchg.h, unless the code is in arch/ and can take care of itself.*

Spinlocks and rwlocks have memory barrier expectations as well. The rule to follow is simple:

1. When acquiring a lock, the implementation must make it globally visible before any subsequent memory operation.
2. When releasing a lock, the implementation must make it such that all previous memory operations are globally visible before the lock release.

Which finally brings us to _atomic_dec_and_lock(). There is an architecture-neutral version implemented in lib/dec_and_lock.c, but most platforms will wish to optimize this in assembler.

```
int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock);
```

Atomically decrement the given counter, and if will drop to zero atomically acquire the given spinlock and perform the decrement of the counter to zero. If it does not drop to zero, do nothing with the spinlock.

It is actually pretty simple to get the memory barrier correct. Simply satisfy the spinlock grab requirements, which is make sure the spinlock operation is globally visible before any subsequent memory operation.

We can demonstrate this operation more clearly if we define an abstract atomic operation:

```
long cas(long *mem, long old, long new);
```

"cas" stands for "compare and swap". It atomically:

1. Compares "old" with the value currently at "mem".
2. If they are equal, "new" is written to "mem".
3. Regardless, the current value at "mem" is returned.

As an example usage, here is what an atomic counter update might look like:

```
void example_atomic_inc(long *counter)
{
        long old, new, ret;

        while (1) {
                old = *counter;
                new = old + 1;

                ret = cas(counter, old, new);
                if (ret == old)
                        break;
        }
}
```

Let's use cas() in order to build a pseudo-C atomic_dec_and_lock():

```
int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock)
{
        long old, new, ret;
        int went_to_zero;

        went_to_zero = 0;
        while (1) {
                old = atomic_read(atomic);
                new = old - 1;
                if (new == 0) {
                        went_to_zero = 1;
                        spin_lock(lock);
                }
                ret = cas(atomic, old, new);
                if (ret == old)
                        break;
                if (went_to_zero) {
                        spin_unlock(lock);
                        went_to_zero = 0;
                }
        }

        return went_to_zero;
}
```

Now, as far as memory barriers go, as long as spin_lock() strictly orders all subsequent memory operations (including the cas()) with respect to itself, things will be fine.

Said another way, _atomic_dec_and_lock() must guarantee that a counter dropping to zero is never made visible before the spinlock being acquired.

Note:

Note that this also means that for the case where the counter is not dropping to zero, there are no memory ordering requirements.

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(struct vm_area_struct *vma, unsigned long address, pte_t *ptep)`

   At the end of every page fault, this routine is invoked to tell the architecture specific code that a translation now exists at virtual address "address" for address space "vma->vm_mm", in the software page tables.

   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.

6. `void tlb_migrate_finish(struct mm_struct *mm)`

   This interface is called at the end of an explicit process migration. This interface provides a hook to allow a platform to update TLB or context-specific information for the address space.

   The ia64 sn2 platform is one example of a platform that uses this interface.

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 map_vm_area() has installed the page table entries. The second is invoked before unmap_kernel_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_page(struct page *page)`

Any time the kernel writes to a page cache page, _OR_ the kernel is about to read from a page cache page and user space shared/writable mappings of this page potentially exist, this routine is called.

> **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 page->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 page->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_page and update_mmu_cache implementations for an example of how to go about doing this.

The idea is, first at flush_dcache_page() time, if page->mapping->i_mmap is an empty tree, just mark the architecture private page flag bit. Later, in update_mmu_cache(), 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_page() 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_kernel_dcache_page(struct page *page)`

When the kernel needs to modify a user page is has obtained with kmap, it calls this function after all modifications are complete (but before kunmapping it) to bring the underlying page up to date. It is assumed here that the user has no incoherent cached copies (i.e. the original page was obtained from a mechanism like get_user_pages()). The default implementation is a nop and should remain so on all coherent architectures. On incoherent architectures, this should flush the kernel cache for page (using page_address(page)).

`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_page and update_mmu_cache. 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.

refcount_t API compared to atomic_t

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.

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()`.

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.

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) - 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()`
- 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 6) - 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

CPU hotplug in the Kernel

Date December, 2016

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>

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.

Command Line Switches

`maxcpus=n` Restrict boot time CPUs to *n*. Say if you have fourV 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 CPUs the kernel will support. If the number supplied here is lower than the number of physically available CPUs than 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.

`cede_offline={"off","on"}` Use this option to disable/enable putting offlined processors to an extended `H_CEDE` state on supported pseries platforms. If nothing is specified, `cede_offline` is set to "on".

This option is limited to the PowerPC architecture.

`cpu0_hotplug` Allow to shutdown CPU0.

This option is limited to the X86 architecture.

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.

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. CPU0 is often special and excluded from CPU hotplug. On X86 the kernel option *CONFIG_BOOTPARAM_HOTPLUG_CPU0* has to be enabled in order to be able to shutdown CPU0. Alternatively the kernel command option *cpu0_hotplug* can be used. Some known dependencies of CPU0:

- Resume from hibernate/suspend. Hibernate/suspend will fail if CPU0 is offline.
- PIC interrupts. CPU0 can't be removed if a PIC interrupt is detected.

Please let Fenghua Yu <fenghua.yu@intel.com> know if you find any dependencies on CPU0.

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.

Using the hotplug API

It is possible to receive notifications once a CPU is offline or onlined. This might be important to certain drivers which need to perform some kind of setup or clean up functions based on the number of available CPUs:

```
#include <linux/cpuhotplug.h>

ret = cpuhp_setup_state(CPUHP_AP_ONLINE_DYN, "X/Y:online",
                        Y_online, Y_prepare_down);
```

X is the subsystem and *Y* the particular driver. The *Y_online* callback will be invoked during registration on all online CPUs. If an error occurs during the online callback the *Y_prepare_down* callback will be invoked on all CPUs on which the online callback was previously invoked. After registration completed, the *Y_online* callback will be invoked once a CPU is brought online and *Y_prepare_down* will be invoked when a CPU is shutdown. All resources which were previously allocated in *Y_online* should be released in *Y_prepare_down*. The return value *ret* is negative if an error occurred during the registration process. Otherwise a positive value is returned which contains the allocated hotplug for dynamically allocated states (*CPUHP_AP_ONLINE_DYN*). It will return zero for predefined states.

The callback can be remove by invoking `cpuhp_remove_state()`. In case of a dynamically allocated state (*CPUHP_AP_ONLINE_DYN*) use the returned state. During the removal of a hotplug state the teardown callback will be invoked.

Multiple instances

If a driver has multiple instances and each instance needs to perform the callback independently then it is likely that a ''multi-state" should be used. First a multi-state state needs to be registered:

```
ret = cpuhp_setup_state_multi(CPUHP_AP_ONLINE_DYN, "X/Y:online,
                              Y_online, Y_prepare_down);
Y_hp_online = ret;
```

The `cpuhp_setup_state_multi()` behaves similar to `cpuhp_setup_state()` except it prepares the callbacks for a multi state and does not invoke the callbacks. This is a one time setup. Once a new instance is allocated, you need to register this new instance:

```
ret = cpuhp_state_add_instance(Y_hp_online, &d->node);
```

This function will add this instance to your previously allocated *Y_hp_online* state and invoke the previously registered callback (*Y_online*) on all online CPUs. The *node* element is a `struct hlist_node` member of your per-instance data structure.

On removal of the instance: :: cpuhp_state_remove_instance(Y_hp_online, &d->node)

should be invoked which will invoke the teardown callback on all online CPUs.

Manual setup

Usually it is handy to invoke setup and teardown callbacks on registration or removal of a state because usually the operation needs to performed once a CPU goes online (offline) and during initial setup (shutdown) of the driver. However each registration and removal function is also available with a `_nocalls` suffix which does not invoke the provided callbacks if the invocation of the callbacks is not desired. During the manual setup (or teardown) the functions `get_online_cpus()` and `put_online_cpus()` should be used to inhibit CPU hotplug operations.

The ordering of the events

The hotplug states are defined in `include/linux/cpuhotplug.h`:

- The states *CPUHP_OFFLINE* … *CPUHP_AP_OFFLINE* are invoked before the CPU is up.
- The states *CPUHP_AP_OFFLINE* … *CPUHP_AP_ONLINE* are invoked just the after the CPU has been brought up. The interrupts are off and the scheduler is not yet active on this CPU. Starting with *CPUHP_AP_OFFLINE* the callbacks are invoked on the target CPU.
- The states between *CPUHP_AP_ONLINE_DYN* and *CPUHP_AP_ONLINE_DYN_END* are reserved for the dynamic allocation.
- The states are invoked in the reverse order on CPU shutdown starting with *CPUHP_ONLINE* and stopping at *CPUHP_OFFLINE*. Here the callbacks are invoked on the CPU that will be shutdown until *CPUHP_AP_OFFLINE*.

A dynamically allocated state via *CPUHP_AP_ONLINE_DYN* is often enough. However if an earlier invocation during the bring up or shutdown is required then an explicit state should be acquired. An explicit state might also be required if the hotplug event requires specific ordering in respect to another hotplug event.

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 callbac 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.

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.

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.

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 callbacks

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 (*)(unsigned int cpu) startup startup callback function

int (*)(unsigned int cpu) teardown teardown callback function

Description

Installs the callback functions and invokes the startup callback on the present cpus which have already reached the **state**.

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 callbacks

Parameters

enum cpuhp_state state The state for which the calls are installed

const char * name Name of the callback.

int (*)(unsigned int cpu) startup startup callback function

int (*)(unsigned int cpu) teardown teardown callback function

Description

Same as **cpuhp_setup_state** except that no calls are executed are invoked during installation of this callback. NOP if SMP=n or HOTPLUG_CPU=n.

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 (*)(unsigned int cpu, struct hlist_node *node) startup startup callback function

int (*)(unsigned int cpu, struct hlist_node *node) teardown teardown callback function

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 startup callback on the present 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**.

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 present cpus which have already reached the **state**.

void **cpuhp_remove_state_nocalls**(enum cpuhp_state *state*)
Remove hotplug state callbacks without invoking teardown

Parameters

enum cpuhp_state state The state for which the calls are removed

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 present cpus which have already reached the **state**.

int **cpuhp_state_remove_instance_nocalls**(enum cpuhp_state *state*, struct hlist_node * *node*)
Remove hotplug instance from state without invoking the reatdown 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.

ID Allocation

Author Matthew Wilcox

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.

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).

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_simple_get()*. To free an ID, call *ida_simple_remove()*.

If you have more complex locking requirements, use a loop around `ida_pre_get()` and *ida_get_new()* to allocate a new ID. Then use *ida_remove()* to free an ID. You must make sure that *ida_get_new()* and *ida_remove()* cannot be called at the same time as each other for the same IDA.

You can also use *ida_get_new_above()* if you need an ID to be allocated above a particular number. *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.

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.

Functions and structures

IDR_INIT(*name*)
Initialise an IDR.

Parameters

name Name of IDR.

Description

A freshly-initialised IDR contains no IDs.

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).

idr sync

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).

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*, *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*, *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_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.

int **ida_get_new**(struct ida * *ida*, int * *p_id*)
    allocate new ID

Parameters

struct ida * ida idr handle

int * p_id pointer to the allocated handle

Description

Simple wrapper around *`ida_get_new_above()`* w/ **starting_id** of zero.

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 **nextid** 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 (*)(int id, void *p, void *data) fn 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**(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_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_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.

IDA description

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_simple_get()*. To free an ID, call *ida_simple_remove()*.

If you have more complex locking requirements, use a loop around `ida_pre_get()` and *ida_get_new()* to allocate a new ID. Then use *ida_remove()* to free an ID. You must make sure that *ida_get_new()* and *ida_remove()* cannot be called at the same time as each other for the same IDA.

You can also use *ida_get_new_above()* if you need an ID to be allocated above a particular number. *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.

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.

int **ida_get_new_above**(struct ida * *ida*, int *start*, int * *id*)
allocate new ID above or equal to a start id

Parameters

struct ida * ida ida handle

int start id to start search at

int * id pointer to the allocated handle

Description

Allocate new ID above or equal to **start**. It should be called with any required locks to ensure that concurrent calls to *ida_get_new_above()* / *ida_get_new()* / *ida_remove()* are not allowed. Consider using *ida_simple_get()* if you do not have complex locking requirements.

If memory is required, it will return `-EAGAIN`, you should unlock and go back to the `ida_pre_get()` call. If the ida is full, it will return `-ENOSPC`. On success, it will return 0.

id returns a value in the range **start** … `0x7fffffff`.

void **ida_remove**(struct ida * *ida*, int *id*)
Free the given ID

Parameters

struct ida * ida ida handle

int id ID to free

Description

This function should not be called at the same time as *ida_get_new_above()*.

void **ida_destroy**(struct ida * *ida*)
Free the contents of an ida

Parameters

struct ida * ida ida handle

Description

Calling this function releases all resources associated with an IDA. When this call returns, the IDA is empty and can be reused or freed. The caller should not allow *ida_remove()* or *ida_get_new_above()* to be called at the same time.

int **ida_simple_get**(struct ida * *ida*, unsigned int *start*, unsigned int *end*, gfp_t *gfp_mask*)
get a new id.

Parameters

struct ida * ida the (initialized) ida.

unsigned int start the minimum id (inclusive, < 0x8000000)

unsigned int end the maximum id (exclusive, < 0x8000000 or 0)

gfp_t gfp_mask memory allocation flags

Description

Allocates an id in the range start <= id < end, or returns -ENOSPC. On memory allocation failure, returns -ENOMEM.

Compared to *ida_get_new_above()* this function does its own locking, and should be used unless there are special requirements.

Use *ida_simple_remove()* to get rid of an id.

void **ida_simple_remove**(struct ida * *ida*, unsigned int *id*)
remove an allocated id.

Parameters

struct ida * ida the (initialized) ida.

unsigned int id the id returned by ida_simple_get.

Description

Use to release an id allocated with *ida_simple_get()*.

Compared to *ida_remove()* this function does its own locking, and should be used unless there are special requirements.

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.

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.

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;
```

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.

How to use local atomic operations

```
#include <linux/percpu.h>
#include <asm/local.h>

static DEFINE_PER_CPU(local_t, counters) = LOCAL_INIT(0);
```

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));
```

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)
{
```

```
        del_timer_sync(&test_timer);
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mathieu Desnoyers");
MODULE_DESCRIPTION("Local Atomic Ops");
```

Concurrency Managed Workqueue (cmwq)

Date September, 2010

Author Tejun Heo <tj@kernel.org>

Author Florian Mickler <florian@mickler.org>

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.

Why cmwq?

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.

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.

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.

Note that the flag WQ_NON_REENTRANT no longer exists as all workqueues are now non-reentrant - any work item is guaranteed to be executed by at most one worker system-wide at any given time.

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.

Currently, for a bound wq, the maximum limit for @max_active is 512 and the default value used when 0 is specified is 256. For an unbound wq, the limit is higher of 512 and 4 * num_possible_cpus(). 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_queue()` should be used to achieve system-wide ST behavior.

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
```

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.

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/debug/tracing/set_event
$ cat /sys/kernel/debug/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.

Kernel Inline Documentations Reference

struct **workqueue_attrs**
A struct for workqueue attributes.

Definition

```
struct workqueue_attrs {
  int nice;
  cpumask_var_t cpumask;
  bool no_numa;
};
```

Members

nice nice level

cpumask allowed CPUs

no_numa disable NUMA affinity

Unlike other fields, no_numa isn't a property of a worker_pool. It only modifies how apply_workqueue_attrs() select pools and thus doesn't participate in pool hash calculations or equality comparisons.

Description

This can be used to change attributes of an unbound workqueue.

work_pending(*work*)
Find out whether a work item is currently pending

Parameters

work The work item in question

delayed_work_pending(*w*)
Find out whether a delayable work item is currently pending

Parameters

w The work item in question

alloc_workqueue(*fmt*, *flags*, *max_active*, *args...*)
allocate a workqueue

Parameters

fmt printf format for the name of the workqueue

flags WQ_* flags

max_active max in-flight work items, 0 for default

args... args for **fmt**

Description

Allocate a workqueue with the specified parameters. For detailed information on WQ_* flags, please refer to Documentation/core-api/workqueue.rst.

The __lock_name macro dance is to guarantee that single lock_class_key doesn't end up with different namesm, which isn't allowed by lockdep.

Return

Pointer to the allocated workqueue on success, NULL on failure.

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.

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.

void **flush_scheduled_work**(void)
ensure that any scheduled work has run to completion.

Parameters

void no arguments

Description

Forces execution of the kernel-global workqueue and blocks until its completion.

Think twice before calling this function! It's very easy to get into trouble if you don't take great care. Either of the following situations will lead to deadlock:

> One of the work items currently on the workqueue needs to acquire a lock held by your code or its caller.
>
> Your code is running in the context of a work routine.

They will be detected by lockdep when they occur, but the first might not occur very often. It depends on what work items are on the workqueue and what locks they need, which you have no control over.

In most situations flushing the entire workqueue is overkill; you merely need to know that a particular work item isn't queued and isn't running. In such cases you should use `cancel_delayed_work_sync()` or `cancel_work_sync()` instead.

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.

Linux generic IRQ handling

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.

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.

Known Bugs And Assumptions

None (knock on wood).

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()`
- *`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 (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.

__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.

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.

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_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_ack_set_bit**(struct *irq_data* * *d*)
Ack pending interrupt via setting bit

Parameters

struct irq_data * d irq_data

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->:c:func:*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.

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;
  cpumask_var_t affinity;
#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 {
  struct device   *parent_device;
  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);
  void (*irq_cpu_online)(struct irq_data *data);
  void (*irq_cpu_offline)(struct irq_data *data);
  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);
  unsigned long   flags;
};
```

Members

parent_device pointer to parent device for irqchip

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

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[0];
};
```

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

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 {
  int pre_vectors;
  int post_vectors;
};
```

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

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.

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

Description

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.

void **synchronize_irq**(unsigned int *irq*)
wait for pending IRQ handlers (on other CPUs)

Parameters

unsigned int irq interrupt number to wait for

Description

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.

This function may be called - with care - from IRQ context.

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 affitnity changed

Description

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_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.

Description

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

Description

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

Description

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

Description

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.

This function may be called - with care - from IRQ context.

bool **disable_hardirq**(unsigned int *irq*)
disables an irq and waits for hardirq completion

Parameters

unsigned int irq Interrupt to disable

Description

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 **enable_irq**(unsigned int *irq*)
enable handling of an irq

Parameters

unsigned int irq Interrupt to enable

Description

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 !

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

Description

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".

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

int **setup_irq**(unsigned int *irq*, struct *irqaction* * *act*)
setup an interrupt

Parameters

unsigned int irq Interrupt line to setup

struct irqaction * act irqaction for the interrupt

Description

Used to statically setup interrupts in the early boot process.

void **remove_irq**(unsigned int *irq*, struct *irqaction* * *act*)
free an 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.

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

Description

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 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

Description

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

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

Description

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.

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

Description

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

Description

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 **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 storeed

Description

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**

Description

This call sets the internal irqchip state of an interrupt, depending on the value of **which**.

This function should be called with preemption disabled if the interrupt controller has per-cpu registers.

int **irq_set_chip**(unsigned int *irq*, struct *irq_chip* * *chip*)
set the irq chip for an irq

Parameters

unsigned int irq irq number

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

Description

Set the hardware irq controller data for an irq

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

Description

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

Description

Set the MSI descriptor entry 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

Description

Set the hardware irq chip data 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_simple_irq**(struct irq_desc * *desc*)
Simple and software-decoded IRQs.

Parameters

struct irq_desc * desc the interrupt description structure for this irq

Description

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

Description

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

Description

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

Description

Only a single callback will be issued to the chip: an ->:c:func:*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

Description

Interrupt occures 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_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

Description

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 **irq_cpu_online**(void)
Invoke all irq_cpu_online functions.

Parameters

void no arguments

Description

Iterate through all irqs and invoke the chip.:c:func:*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.:c:func:*irq_cpu_offline()* for each.

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

Description

Like *handle_fasteoi_irq()*, but for use with hierarchy where the irq_chip also needs to have its ->:c:func:*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

Description

Like *handle_fasteoi_irq()*, but for use with hierarchy where the irq_chip also needs to have its ->:c:func:*irq_mask_ack()* function called.

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_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

Conditinal, 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_compose_msi_msg**(struct *irq_data* * *data*, struct msi_msg * *msg*)
Componse 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()*.

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

int **__handle_domain_irq**(struct irq_domain * *domain*, unsigned int *hwirq*, bool *lookup*, struct pt_regs * *regs*)
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

bool lookup Whether to perform the domain lookup or not

struct pt_regs * regs Register file coming from the low-level handling code

Return

0 on success, or -EINVAL if conversion has failed

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 cpumask * *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 cpumask * 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_alloc_hwirqs**(int *cnt*, int *node*)
Allocate an irq descriptor and initialize the hardware

Parameters

int cnt number of interrupts to allocate

int node node on which to allocate

Description

Returns an interrupt number > 0 or 0, if the allocation fails.

void **irq_free_hwirqs**(unsigned int *from*, int *cnt*)
Free irq descriptor and cleanup the hardware

Parameters

unsigned int from Free from irq number

int cnt number of interrupts to free

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**(unsigned int *irq*)
Get the statistics for an interrupt

Parameters

unsigned int irq The interrupt number

Description

Returns the sum of interrupt counts on all cpus 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

Parameters

unsigned int irq The interrupt number

Description

Returns the sum of interrupt counts on all cpus since boot for **irq**. Contrary to `kstat_irqs()` this can be called from any preemptible context. It's protected against concurrent removal of an interrupt descriptor when sparse irqs are enabled.

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.

int **irq_set_chip**(unsigned int *irq*, struct irq_chip * *chip*)
set the irq chip for an irq

Parameters

unsigned int irq irq number

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

Description

Set the hardware irq controller data for an irq

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

Description

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

Description

Set the MSI descriptor entry 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

Description

Set the hardware irq chip data 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_simple_irq**(struct irq_desc * *desc*)
Simple and software-decoded IRQs.

Parameters

struct irq_desc * desc the interrupt description structure for this irq

Description

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

Description

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

Description

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

Description

Only a single callback will be issued to the chip: an ->:c:func:*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

Description

Interrupt occures 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_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

Description

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 **irq_cpu_online**(void)
Invoke all irq_cpu_online functions.

Parameters

void no arguments

Description

Iterate through all irqs and invoke the chip.:c:func:*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.:c:func:*irq_cpu_offline()* for each.

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

Description

Like *handle_fasteoi_irq()*, but for use with hierarchy where the irq_chip also needs to have its ->:c:func:*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

Description

Like *handle_fasteoi_irq()*, but for use with hierarchy where the irq_chip also needs to have its ->:c:func:*irq_mask_ack()* function called.

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_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

Conditinal, 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_compose_msi_msg**(struct *irq_data* * *data*, struct msi_msg * *msg*)
Componse 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()*.

Credits

The following people have contributed to this document:

1. Thomas Gleixner tglx@linutronix.de
2. Ingo Molnar mingo@elte.hu

Using flexible arrays in the kernel

Large contiguous memory allocations can be unreliable in the Linux kernel. Kernel programmers will sometimes respond to this problem by allocating pages with *vmalloc()*. This solution not ideal, though. On 32-bit systems, memory from vmalloc() must be mapped into a relatively small address space; it's easy to run out. On SMP systems, the page table changes required by vmalloc() allocations can require expensive cross-processor interrupts on all CPUs. And, on all systems, use of space in the vmalloc() range increases pressure on the translation lookaside buffer (TLB), reducing the performance of the system.

In many cases, the need for memory from vmalloc() can be eliminated by piecing together an array from smaller parts; the flexible array library exists to make this task easier.

A flexible array holds an arbitrary (within limits) number of fixed-sized objects, accessed via an integer index. Sparse arrays are handled reasonably well. Only single-page allocations are made, so memory allocation failures should be relatively rare. The down sides are that the arrays cannot be indexed directly, individual object size cannot exceed the system page size, and putting data into a flexible array requires a copy operation. It's also worth noting that flexible arrays do no internal locking at all; if concurrent access to an array is possible, then the caller must arrange for appropriate mutual exclusion.

The creation of a flexible array is done with *flex_array_alloc()*:

```
#include <linux/flex_array.h>

struct flex_array *flex_array_alloc(int element_size,
                                    unsigned int total,
                                    gfp_t flags);
```

The individual object size is provided by `element_size`, while total is the maximum number of objects which can be stored in the array. The flags argument is passed directly to the internal memory allocation calls. With the current code, using flags to ask for high memory is likely to lead to notably unpleasant side effects.

It is also possible to define flexible arrays at compile time with:

```
DEFINE_FLEX_ARRAY(name, element_size, total);
```

This macro will result in a definition of an array with the given name; the element size and total will be checked for validity at compile time.

Storing data into a flexible array is accomplished with a call to *flex_array_put()*:

```
int flex_array_put(struct flex_array *array, unsigned int element_nr,
                   void *src, gfp_t flags);
```

This call will copy the data from src into the array, in the position indicated by `element_nr` (which must be less than the maximum specified when the array was created). If any memory allocations must be performed, flags will be used. The return value is zero on success, a negative error code otherwise.

There might possibly be a need to store data into a flexible array while running in some sort of atomic context; in this situation, sleeping in the memory allocator would be a bad thing. That can be avoided by using `GFP_ATOMIC` for the flags value, but, often, there is a better way. The trick is to ensure that any needed memory allocations are done before entering atomic context, using *`flex_array_prealloc()`*:

```
int flex_array_prealloc(struct flex_array *array, unsigned int start,
                        unsigned int nr_elements, gfp_t flags);
```

This function will ensure that memory for the elements indexed in the range defined by `start` and `nr_elements` has been allocated. Thereafter, a `flex_array_put()` call on an element in that range is guaranteed not to block.

Getting data back out of the array is done with *`flex_array_get()`*:

```
void *flex_array_get(struct flex_array *fa, unsigned int element_nr);
```

The return value is a pointer to the data element, or NULL if that particular element has never been allocated.

Note that it is possible to get back a valid pointer for an element which has never been stored in the array. Memory for array elements is allocated one page at a time; a single allocation could provide memory for several adjacent elements. Flexible array elements are normally initialized to the value FLEX_ARRAY_FREE (defined as 0x6c in <linux/poison.h>), so errors involving that number probably result from use of unstored array entries. Note that, if array elements are allocated with __GFP_ZERO, they will be initialized to zero and this poisoning will not happen.

Individual elements in the array can be cleared with *`flex_array_clear()`*:

```
int flex_array_clear(struct flex_array *array, unsigned int element_nr);
```

This function will set the given element to FLEX_ARRAY_FREE and return zero. If storage for the indicated element is not allocated for the array, `flex_array_clear()` will return `-EINVAL` instead. Note that clearing an element does not release the storage associated with it; to reduce the allocated size of an array, call *`flex_array_shrink()`*:

```
int flex_array_shrink(struct flex_array *array);
```

The return value will be the number of pages of memory actually freed. This function works by scanning the array for pages containing nothing but FLEX_ARRAY_FREE bytes, so (1) it can be expensive, and (2) it will not work if the array's pages are allocated with __GFP_ZERO.

It is possible to remove all elements of an array with a call to *`flex_array_free_parts()`*:

```
void flex_array_free_parts(struct flex_array *array);
```

This call frees all elements, but leaves the array itself in place. Freeing the entire array is done with *`flex_array_free()`*:

```
void flex_array_free(struct flex_array *array);
```

As of this writing, there are no users of flexible arrays in the mainline kernel. The functions described here are also not exported to modules; that will probably be fixed when somebody comes up with a need for it.

Flexible array functions

struct flex_array * **flex_array_alloc**(int *element_size*, unsigned int *total*, gfp_t *flags*)
Creates a flexible array.

Parameters

int element_size individual object size.

unsigned int total maximum number of objects which can be stored.

gfp_t flags GFP flags

Return

Returns an object of structure flex_array.

int **flex_array_prealloc**(struct flex_array * *fa*, unsigned int *start*, unsigned int *nr_elements*, gfp_t *flags*)
Ensures that memory for the elements indexed in the range defined by start and nr_elements has been allocated.

Parameters

struct flex_array * fa array to allocate memory to.

unsigned int start start address

unsigned int nr_elements number of elements to be allocated.

gfp_t flags GFP flags

void **flex_array_free**(struct flex_array * *fa*)
Removes all elements of a flexible array.

Parameters

struct flex_array * fa array to be freed.

void **flex_array_free_parts**(struct flex_array * *fa*)
Removes all elements of a flexible array, but leaves the array itself in place.

Parameters

struct flex_array * fa array to be emptied.

int **flex_array_put**(struct flex_array * *fa*, unsigned int *element_nr*, void * *src*, gfp_t *flags*)
Stores data into a flexible array.

Parameters

struct flex_array * fa array where element is to be stored.

unsigned int element_nr position to copy, must be less than the maximum specified when the array was created.

void * src data source to be copied into the array.

gfp_t flags GFP flags

Return

Returns zero on success, a negative error code otherwise.

int **flex_array_clear**(struct flex_array * *fa*, unsigned int *element_nr*)
Clears an individual element in the array, sets the given element to FLEX_ARRAY_FREE.

Parameters

struct flex_array * fa array to which element to be cleared belongs.

unsigned int element_nr element position to clear.

Return

Returns zero on success, -EINVAL otherwise.

void * **flex_array_get**(struct flex_array * *fa*, unsigned int *element_nr*)
Retrieves data into a flexible array.

Parameters

struct flex_array * fa array from which data is to be retrieved.

unsigned int element_nr Element position to retrieve data from.

Return

Returns a pointer to the data element, or NULL if that particular element has never been allocated.

int **flex_array_shrink**(struct flex_array * *fa*)
Reduces the allocated size of an array.

Parameters

struct flex_array * fa array to shrink.

Return

Returns number of pages of memory actually freed.

Reed-Solomon Library Programming Interface

Author Thomas Gleixner

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.

Known Bugs And Assumptions

None.

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);
```

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[0];
};
```

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.

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 **GFP_** flags for allocations

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 (*)(int) gffunc 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)

Description

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 (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

Description

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 bits or -EBADMSG for uncorrectable errors.

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!)

Description

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 (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

Description

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 bits or -EBADMSG for uncorrectable errors.

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

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_virt**(struct gen_pool * *pool*, unsigned long *virt*, phys_addr_t *phys*, 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 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

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.

void **gen_pool_free**(struct gen_pool * *pool*, unsigned long *addr*, size_t *size*)
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

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**(struct gen_pool * *pool*, size_t *size*, genpool_algo_t *algo*, void * *data*)
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

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 (*)(struct gen_pool *pool, struct gen_pool_chunk *chunk, void *data) func 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 **addr_in_gen_pool**(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.

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.

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.

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.

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.

How to get printk format specifiers right

Author Randy Dunlap <rdunlap@infradead.org>

Author Andrew Murray <amurray@mpc-data.co.uk>

Integer types

```
If variable is of Type,         use printk format specifier:
------------------------------------------------------------
        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
        s32                     %d or %x
        u32                     %u or %x
        s64                     %lld or %llx
        u64                     %llu or %llx
```

If <type> is dependent on a config option for its size (e.g., sector_t, blkcnt_t) or is architecture-dependent for its size (e.g., tcflag_t), use a format specifier of its largest possible type and explicitly cast to it.

Example:

```
printk("test: sector number/total blocks: %llu/%llu\n",
        (unsigned long long)sector, (unsigned long long)blockcount);
```

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().

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.

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. If you *really* want the address see %px below.

Symbols/Function Pointers

```
%pS     versatile_init+0x0/0x110
%ps     versatile_init
%pF     versatile_init+0x0/0x110
%pf     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.

Note, that the F and `f` specifiers are identical to S (`s`) and thus deprecated. We have F and `f` because on ia64, ppc64 and parisc64 function pointers are indirect and, in fact, are function descriptors, which require additional dereferencing before we can lookup the symbol. As of now, S and `s` perform dereferencing on those platforms (when needed), so F and `f` exist for compatibility reasons only.

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.

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/sysctl/kernel.txt for more details.

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.

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 http://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 http://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.

kobjects

```
%pOF[fnpPcCF]
```

For printing kobject based structs (device nodes). 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.

struct clk

```
%pC     pll1
%pCn    pll1
```

For printing struct clk structures. %pC and %pCn print the name (Common Clock Framework) or address (legacy clock framework) of the structure.

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.

Passed by reference.

Flags bitfields such as page flags, gfp_flags

```
%pGp    referenced|uptodate|lru|active|private
%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]age flags, [v]ma_flags (both expect `unsigned long *`) and [g]fp_flags (expects `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.

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.

Circular Buffers

Author David Howells <dhowells@redhat.com>

Author Paul E. McKenney <paulmck@linux.vnet.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.

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.

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.

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.

Further reading

See also Documentation/memory-barriers.txt for a description of Linux's memory barrier facilities.

GFP masks used from FS/IO context

Date May, 2018

Author Michal Hocko <mhocko@kernel.org>

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.

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 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 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.

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

TWO

INTERFACES FOR KERNEL DEBUGGING

The object-lifetime debugging infrastructure

Author Thomas Gleixner

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.

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.

Debug functions

void **debug_object_init**(void * *addr*, struct *debug_obj_descr* * *descr*)
debug checks when an object is initialized

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when an object on stack is initialized

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when an object is activated

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when an object is deactivated

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when an object is destroyed

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when an object is freed

Parameters

void * addr address of the object

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*, struct *debug_obj_descr* * *descr*)
debug checks when object should be init-ed

Parameters

void * addr address of the object

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.

Fixup functions

Debug object type description structure

struct **debug_obj**
representaion of an tracked object

Definition

```
struct debug_obj {
  struct hlist_node       node;
  enum debug_obj_state    state;
  unsigned int            astate;
  void *object;
  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.

Known Bugs And Assumptions

None (knock on wood).

The Linux Kernel Tracepoint API

Author Jason Baron

Author William Cohen

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.

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.

SIGNAL

void **trace_signal_generate**(int *sig*, struct siginfo * *info*, struct task_struct * *task*, int *group*, int *result*)
called when a signal is generated

Parameters

int sig signal number

struct 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 siginfo * *info*, struct k_sigaction * *ka*)
called when a signal is delivered

Parameters

int sig signal number

struct 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.

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_queue * *q*, struct request * *rq*)
place block IO request back on a queue

Parameters

struct request_queue * q queue holding operation

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*, int *error*, unsigned int *nr_bytes*)
block IO operation completed by device driver

Parameters

struct request * rq block operations request

int 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_insert**(struct request_queue * *q*, struct request * *rq*)
insert block operation request into queue

Parameters

struct request_queue * q target queue

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_queue * *q*, struct request * *rq*)
issue pending block IO request operation to device driver

Parameters

struct request_queue * q queue holding operation

struct request * rq block IO operation operation request

Description

Called when block operation request **rq** from queue **q** is sent to a device driver for processing.

void **trace_block_bio_bounce**(struct request_queue * *q*, struct bio * *bio*)
used bounce buffer when processing block operation

Parameters

struct request_queue * q queue holding the block operation

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_complete**(struct request_queue * *q*, struct bio * *bio*, int *error*)
completed all work on the block operation

Parameters

struct request_queue * q queue holding the block operation

struct bio * bio block operation completed

int error io error value

Description

This tracepoint indicates there is no further work to do on this block IO operation **bio**.

void **trace_block_bio_backmerge**(struct request_queue * *q*, struct request * *rq*, struct bio * *bio*)
merging block operation to the end of an existing operation

Parameters

struct request_queue * q queue holding operation

struct request * rq request bio is being merged into

struct bio * bio new block operation to merge

Description

Merging block request **bio** to the end of an existing block request in queue **q**.

void **trace_block_bio_frontmerge**(struct request_queue * *q*, struct request * *rq*, struct bio * *bio*)
merging block operation to the beginning of an existing operation

Parameters

struct request_queue * q queue holding operation

struct request * rq request bio is being merged into

struct bio * bio new block operation to merge

Description

Merging block IO operation **bio** to the beginning of an existing block operation in queue **q**.

void **trace_block_bio_queue**(struct request_queue * *q*, struct bio * *bio*)
putting new block IO operation in queue

Parameters

struct request_queue * q queue holding operation

struct bio * bio new block operation

Description

About to place the block IO operation **bio** into queue **q**.

void **trace_block_getrq**(struct request_queue * *q*, struct bio * *bio*, int *rw*)
get a free request entry in queue for block IO operations

Parameters

struct request_queue * q queue for operations

struct bio * bio pending block IO operation (can be NULL)

int rw low bit indicates a read (0) or a write (1)

Description

A request struct for queue **q** has been allocated to handle the block IO operation **bio**.

void **trace_block_sleeprq**(struct request_queue * *q*, struct bio * *bio*, int *rw*)
waiting to get a free request entry in queue for block IO operation

Parameters

struct request_queue * q queue for operation

struct bio * bio pending block IO operation (can be NULL)

int rw low bit indicates a read (0) or a write (1)

Description

In the case where a request struct cannot be provided for queue **q** the process needs to wait for an request struct to become available. This tracepoint event is generated each time the process goes to sleep waiting for request struct become available.

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 request_queue * *q*, struct bio * *bio*, unsigned int *new_sector*)
split a single bio struct into two bio structs

Parameters

struct request_queue * q queue containing the bio

struct bio * bio block operation being split

unsigned int new_sector The starting sector for the new bio

Description

The bio request **bio** in request queue **q** 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 limitation such as operation crossing device boundaries in a RAID system.

void **trace_block_bio_remap**(struct request_queue * *q*, struct bio * *bio*, dev_t *dev*, sector_t *from*)
map request for a logical device to the raw device

Parameters

struct request_queue * q queue holding the operation

struct bio * bio revised operation

dev_t dev 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_queue * *q*, struct request * *rq*, dev_t *dev*, sector_t *from*)
map request for a block operation request

Parameters

struct request_queue * q queue holding the operation

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.

Workqueue

void **trace_workqueue_queue_work**(unsigned int *req_cpu*, struct pool_workqueue * *pwq*, struct work_struct * *work*)
called when a work gets queued

Parameters

unsigned 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*)
called immediately after the workqueue callback

Parameters

struct work_struct * work pointer to struct work_struct

Description

Allows to track workqueue execution.

Symbols

A

B

C

D

H

I

M

N

O

P

Q

R

S

T

U

V