The Linux Kernel API — The Linux Kernel documentation

Data Types¶

Doubly Linked Lists¶

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/printk-formats.txt 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 * `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**.

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.

Basic Kernel Library Functions¶

The Linux kernel provides more basic utility functions.

Bitmap Operations¶

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]

[1]	(*1*, *2*) For these marked lines, if we hadn't first done `bitmap_fold()` into tmp, then the **dst** result would have been empty.

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

`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).

unsigned int `bitmap_from_u32array` (unsigned long * *bitmap*, unsigned int *nbits*, const u32 * *buf*, unsigned int *nwords*)¶

copy the contents of a u32 array of bits to bitmap

Parameters

`unsigned long * bitmap`

array of unsigned longs, the destination bitmap, non NULL

`unsigned int nbits`

number of bits in **bitmap**

`const u32 * buf`

array of u32 (in host byte order), the source bitmap, non NULL

`unsigned int nwords`

number of u32 words in **buf**

Description

copy min(nbits, 32*nwords) bits from **buf** to **bitmap**, remaining bits between nword and nbits in **bitmap** (if any) are cleared. In last word of **bitmap**, the bits beyond nbits (if any) are kept unchanged.

Return the number of bits effectively copied.

unsigned int `bitmap_to_u32array` (u32 * *buf*, unsigned int *nwords*, 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, non NULL

`unsigned int nwords`

number of u32 words in **buf**

`const unsigned long * bitmap`

array of unsigned longs, the source bitmap, non NULL

`unsigned int nbits`

number of bits in **bitmap**

Description

copy min(nbits, 32*nwords) bits from **bitmap** to **buf**. Remaining bits after nbits in **buf** (if any) are cleared.

Return the number of bits effectively copied.

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.

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**.

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.

CRC Functions¶

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.

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.

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

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 for the data buffer

Parameters

`u16 crc`
previous CRC value

`u8 const * buffer`
data pointer

`size_t len`
: number of bytes in the buffer

idr/ida Functions¶

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).

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 `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 [start, end). Returns -ENOSPC if there are no unused IDs in that range.

Note that **end** is treated as max when <= 0. This is to always allow using **start** + N as **end** as long as N is inside integer range.

Simultaneous modifications to the **idr** are not allowed and should be prevented by the user, usually with a lock. `idr_alloc()` may be called concurrently with read-only accesses to the **idr**, such as `idr_find()` and `idr_for_each_entry()`.

int `idr_alloc_cyclic` (struct idr * *idr*, void * *ptr*, int *start*, int *end*, gfp_t *gfp*)¶

allocate new idr entry in a cyclical fashion

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 ID larger than the last ID allocated if one is available. If not, it will attempt to allocate the smallest ID that is larger or equal to **start**.

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 pointer and the data pointer passed to this function.

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 lowest possible ID to return

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*, int *id*)¶

replace pointer for given id

Parameters

`struct idr * idr`
idr handle

`void * ptr`
New pointer to associate with the ID

`int id`
Lookup key

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

0 on success. `-ENOENT` indicates that **id** was not found. `-EINVAL` indicates that **id** or **ptr** were not valid.

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.

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_COLD` - Request cache-cold pages instead of
trying to return cache-warm pages.

`__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_REPEAT` - If allocation fails initially, try once more before failing.

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

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

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.

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_REPEAT is supported only for large (>32kB) allocations, and it should be used only if kmalloc is preferable to the vmalloc fallback, due to visible performance drawbacks.

Any use of gfp flags outside of GFP_KERNEL should be consulted with mm people.

User Space Memory Access¶

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.

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 `filemap_fdatawait` (struct address_space * *mapping*)¶

wait for all under-writeback pages to complete

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. 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 `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 `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_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_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_tag` (struct address_space * *mapping*, pgoff_t * *index*, int *tag*, unsigned int *nr_pages*, struct page ** *pages*)¶

find and return pages that match **tag**

Parameters

`struct address_space * mapping`
the address_space to search

`pgoff_t * index`
the starting page index

`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.

int `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).

int `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.

Returns 0 if successful.

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 page 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, however it's the caller's responsibility to guarantee that we are the only thread that changes migrate type of pageblocks the pages fall in.

The PFN range must belong to a single zone.

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_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.

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*, int *wait*)¶

write out a single page and optionally wait on I/O

Parameters

`struct page * page`

the page to write

`int wait`

if true, wait on writeout

Description

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

`write_one_page()` returns a negative error code if I/O failed.

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.

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

initialise ipc identifiers

Parameters

`struct ipc_ids * ids`
ipc identifier set

Description

Set up the sequence range to use for the ipc identifier range (limited below IPCMNI) then initialise the 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 ipc_ids.rwsem held.

int `ipc_get_maxid` (struct ipc_ids * *ids*)¶

get the last assigned id

Parameters

`struct ipc_ids * ids`
ipc identifier set

Description

Called with ipc_ids.rwsem held.

int `ipc_addid` (struct ipc_ids * *ids*, struct kern_ipc_perm * *new*, int *size*)¶

add an ipc identifier

Parameters

`struct ipc_ids * ids`
ipc identifier set
`struct kern_ipc_perm * new`
new ipc permission set
`int size`
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_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_alloc` (int *size*)¶

allocate ipc space

Parameters

`int size`
size desired

Description

Allocate memory from the appropriate pools and return a pointer to it. NULL is returned if the allocation fails

void `ipc_free` (void * *ptr*)¶

free ipc space

Parameters

`void * ptr`
pointer returned by ipc_alloc

Description

Free a block created with `ipc_alloc()` .

void * `ipc_rcu_alloc` (int *size*)¶

allocate ipc and rcu space

Parameters

`int size`
size desired

Description

Allocate memory for the rcu header structure + the object. Returns the pointer to the object or NULL upon failure.

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 numer 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 initialize a fifo using a preallocated buffer.

The numer 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.

unsigned int `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.

struct subprocess_info * `call_usermodehelper_setup` (const char * *path*, char ** *argv*, char ** *envp*, gfp_t *gfp_mask*, int (*init) (struct subprocess_info **info*, struct cred **new*, void (*cleanup) (struct subprocess_info **info*, void * *data*)¶

prepare to call a usermode helper

Parameters

`const char * path`
path to usermode executable

`char ** argv`
arg vector for process

`char ** envp`
environment for process

`gfp_t gfp_mask`
gfp mask for memory allocation

`int (*)(struct subprocess_info *info, struct cred *new) init`
an init function

`void (*)(struct subprocess_info *info) cleanup`
a cleanup function

`void * data`
: arbitrary context sensitive data

Description

Returns either `NULL` on allocation failure, or a subprocess_info structure. This should be passed to call_usermodehelper_exec to exec the process and free the structure.

The init function is used to customize the helper process prior to exec. A non-zero return code causes the process to error out, exit, and return the failure to the calling process

The cleanup function is just before ethe subprocess_info is about to be freed. This can be used for freeing the argv and envp. The Function must be runnable in either a process context or the context in which call_usermodehelper_exec is called.

int `call_usermodehelper_exec` (struct subprocess_info * *sub_info*, int *wait*)¶

: start a usermode application

Parameters

`struct subprocess_info * sub_info`
: information about the subprocessa

`int wait`
: wait for the application to finish and return status. when UMH_NO_WAIT don’t wait at all, but you get no useful error back when the program couldn’t be exec’ed. This makes it safe to call from interrupt context.

Description

Runs a user-space application. The application is started asynchronously if wait is not set, and runs as a child of system workqueues. (ie. it runs with full root capabilities and optimized affinity).

int `call_usermodehelper` (const char * *path*, char ** *argv*, char ** *envp*, int *wait*)¶

: prepare and start a usermode application

Parameters

`const char * path`
: path to usermode executable

`char ** argv`
: arg vector for process

`char ** envp`
: environment for process

`int wait`
: wait for the application to finish and return status. when UMH_NO_WAIT don’t wait at all, but you get no useful error back when the program couldn’t be exec’ed. This makes it safe to call from interrupt context.

Description

This function is the equivalent to use `call_usermodehelper_setup()` and `call_usermodehelper_exec()`.

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

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*, 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.

`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 is the basic “create a file” function for securityfs. It allows for a wide range of flexibility in creating a file, or a directory (if you want to create a directory, the `securityfs_create_dir()` function is recommended to be used instead).

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.

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.

void `audit_log_secctx` (struct audit_buffer * *ab*, u32 *secid*)¶

Converts and logs SELinux context

Parameters

`struct audit_buffer * ab`

audit_buffer

`u32 secid`

security number

Description

This is a helper function that calls security_secid_to_secctx to convert secid to secctx and then adds the (converted) SELinux context to the audit log by calling audit_log_format, thus also preventing leak of internal secid to userspace. If secid cannot be converted audit_panic is called.

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 timespec * *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 timespec * 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.

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_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. Queue lock must be held.

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()` . Queue lock must be held.

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. Queue lock must be held.

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.

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**. This variant must be called with the queue lock held and interrupts disabled.

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. The caller must hold the queue lock.

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.

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_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).

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`(int *cpu*, struct hd_struct * *part*)¶

Round off the performance stats on a struct disk_stats.

Parameters

`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 `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.

int `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.

Context

queue_lock must be held.

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.

Context

queue_lock must be held.

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.
>
> Block internal functions which don't want to start timer should call `blk_dequeue_request()`.

Context

queue_lock must be held.

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.

Context

queue_lock must be held.

bool `blk_update_request` (struct request * *req*, int *error*, unsigned int *nr_bytes*)¶

Special helper function for request stacking drivers

Parameters

`struct request * req`
the request being processed
`int error`
`0` for success, < `0` for error
`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*, int *error*, unsigned int *nr_bytes*)¶

Helper function for drivers to complete the request.

Parameters

`struct request * rq`
the request being processed
`int error`
`0` for success, < `0` for error
`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*, int *error*)¶

Helper function for drives to finish the request.

Parameters

`struct request * rq`
the request to finish
`int error`
`0` for success, < `0` for error

Description

> Completely finish **rq**.

bool `__blk_end_request` (struct request * *rq*, int *error*, unsigned int *nr_bytes*)¶

Helper function for drivers to complete the request.

Parameters

`struct request * rq`
the request being processed
`int error`
`0` for success, < `0` for error
`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*, int *error*)¶

Helper function for drives to finish the request.

Parameters

`struct request * rq`
the request to finish
`int error`
`0` for success, < `0` for error

Description

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

bool `__blk_end_request_cur` (struct request * *rq*, int *error*)¶

Helper function to finish the current request chunk.

Parameters

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

`int error`
`0` for success, < `0` for error

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.

struct request * `__get_request` (struct request_list * *rl*, unsigned int *op*, struct bio * *bio*, 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`)

`gfp_t gfp_mask`
: allocation mask

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*, gfp_t *gfp_mask*)¶

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`)

`gfp_t gfp_mask`
: allocation mask

Description

Get a free request from **q**. If `__GFP_DIRECT_RECLAIM` is set in **gfp_mask**, 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*, int *error*, unsigned int *nr_bytes*, unsigned int *bidi_bytes*)¶

Complete a bidi request

Parameters

`struct request * rq`
: the request to complete

`int error`
: `0` for success, < `0` for error

`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*, int *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

`int error`
: `0` for success, < `0` for error

`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_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 `__bio_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.
>
> Note that this function may fail with -EOPNOTSUPP if the driver signals zeroing offload support, but the device fails to process the command (for some devices there is no non-destructive way to verify whether this operation is actually supported). In this case the caller should call retry the call to `blkdev_issue_zeroout()` and the fallback path will be used.
>
> 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.

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.

Notes

> queue lock must be 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.

Notes

> queue lock must be held.

void `blk_queue_invalidate_tags` (struct request_queue * *q*)¶

: invalidate all pending tags

Parameters

`struct request_queue * q`
: the request queue for the device

Description

> Hardware conditions may dictate a need to stop all pending requests. In this case, we will safely clear the block side of the tag queue and readd all requests to the request queue in the right order.

Notes

> queue lock must be held.

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.

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*)¶

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

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 `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_mutx locked.

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, 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..255]. 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..255] 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..255] or a negative error code otherwise

void `device_add_disk` (struct device * *parent*, struct gendisk * *disk*)¶

add partitioning information to kernel list

Parameters

`struct device * parent`
parent device for the disk

`struct gendisk * disk`
per-device partitioning information

Description

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

FIXME: error handling

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).

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.

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_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.

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.

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 `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.

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.
