Linux Memeory Manager

Generated by Doxygen 1.10.0

Chapter 1

Generating Perfect Hash Functions with gperf

1.1 Overview

This guide explains how to use `gperf` to generate perfect hash functions and lookup tables for efficient keyword lookup in C programs.

1.2 Prerequisites

Before getting started, make sure you have `gperf` installed on your system. You can install it using the package manager of your operating system or by downloading and compiling it from source.

1.3 Usage

1.3.1 1. Prepare Keyword List

Create a text file containing the list of keywords (or data type names) for which you want to generate the perfect hash function. Each keyword should be on a separate line.

Example (`data_type_keywords.txt`):

```
int
char
float
double
short
long
long long
unsigned int
unsigned char
unsigned short
unsigned long
unsigned long long
```

1.3.2 2. Generate Perfect Hash Function

Run `gperf` on the keyword list file to generate the perfect hash function and associated lookup table. Here's the command:

```
gperf -L C -t -N getSizeOfDataType -K name -H hash_function data_type_keywords.txt > data_type_mappings.c
```

Explanation of options:

- `-L C`: Specifies the language to use (C).
- `-t`: Tells `gperf` to produce a C function table.
- `-N getSizeOfDataType`: Specifies the name of the lookup function.
- `-K name`: Specifies the key field in the input file (data type names).
- `-H hash_function`: Specifies the hash function to use.
- `data_type_keywords.txt`: Input file containing the list of data type names.
- `data_type_mappings.c`: Output C file containing the generated code.

1.3.3 3. Include Generated Code

Include the generated C code (`data_type_mappings.c`) in your project. This code contains the perfect hash function and the lookup table for efficient keyword lookup.

1.4 Example

Here's how you can use the generated perfect hash function and lookup table in your C program:

```
#include <stdio.h>
#include "data_type_mappings.c"

size_t getSizeOfDataType(const char *data_type) {
    // Use the generated perfect hash function to get the size of the data type
    const struct DataTypeMapping *result = in_word_set(data_type, strlen(data_type));
    if (result) {
        return result->size;
    }
    return 0; // Return 0 if data type is not found
}

int main() {
    printf("Size of int: %zu\n", getSizeOfDataType("int"));
    printf("Size of float: %zu\n", getSizeOfDataType("float"));
    printf("Size of long long: %zu\n", getSizeOfDataType("long long"));
    return 0;
}
```

1.5 Conclusion

By using `gperf` to generate perfect hash functions and lookup tables, you can efficiently perform keyword lookup in your C programs, improving performance compared to linear search algorithms.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

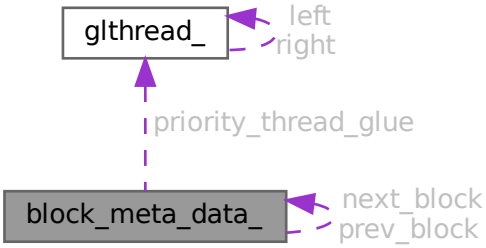
Data Structure Documentation

4.1 block_meta_data_ Struct Reference

Structure representing metadata for a memory block.

`#include <memory_manager.h>`

Collaboration diagram for block_meta_data_:



Data Fields

- vm_bool_t is_free
- uint32_t block_size
- uint32_t offset
- struct block_meta_data_ * prev_block
- struct block_meta_data_ * next_block
- glthread_t priority_thread_glue

4.1.1 Detailed Description

Structure representing metadata for a memory block.

The `block_meta_data_t` structure represents metadata for a memory block. It includes information such as whether the block is free or allocated, its size, pointers to the previous and next blocks (if applicable), and the offset within the memory region.

4.1.2 Field Documentation

4.1.2.1 block_size

```
uint32_t block_meta_data_::block_size
```

Size of the memory block.

4.1.2.2 is_free

```
vm_bool_t block_meta_data_::is_free
```

Flag indicating whether the block is free.

4.1.2.3 next_block

```
struct block_meta_data_* block_meta_data_::next_block
```

Pointer to the next memory block.

4.1.2.4 offset

```
uint32_t block_meta_data_::offset
```

Offset within the memory region.

4.1.2.5 prev_block

```
struct block_meta_data_* block_meta_data_::prev_block
```

Pointer to the previous memory block.

4.1.2.6 priority_thread_glue

```
glthread_t block_meta_data_::priority_thread_glue
```

Priority thread glue for managing block priority.

The documentation for this struct was generated from the following file:

- memory_manager.h

4.2 datatype_mapping_t Struct Reference

Structure representing a data type mapping.

Data Fields

- const char * name
- size_t size

4.2.1 Detailed Description

Structure representing a data type mapping.

This structure defines a mapping between a data type name and its corresponding size.

4.2.2 Field Documentation

4.2.2.1 name

```
const char* datatype_mapping_t::name
```

The name of the data type.

4.2.2.2 size

```
size_t datatype_mapping_t::size
```

The size of the data type in bytes.

The documentation for this struct was generated from the following file:

- datatype_size_lookup.c

4.3 emp_ Struct Reference

Structure representing an employee.

Data Fields

- char name [32]
- uint32_t emp_id

4.3.1 Detailed Description

Structure representing an employee.

This structure defines the attributes of an employee, including their name and employee ID.

4.3.2 Field Documentation

4.3.2.1 emp_id

```
uint32_t emp_::emp_id
```

The employee ID.

4.3.2.2 name

```
char emp_::name[32]
```

The name of the employee.

The documentation for this struct was generated from the following file:

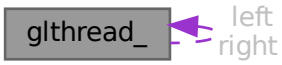
- memory_manager_test.c

4.4 glthread_ Struct Reference

Structure representing a generic linked list node for threaded linking.

```
#include <glthread.h>
```

Collaboration diagram for glthread_:



Data Fields

- struct glthread_ * left
- struct glthread_ * right

4.4.1 Detailed Description

Structure representing a generic linked list node for threaded linking.

This structure defines a generic linked list node for threaded linking. It consists of left and right pointers for threading the nodes together.

4.4.2 Field Documentation

4.4.2.1 left

```
struct glthread_* glthread_::left
```

Pointer to the left node in the linked list.

4.4.2.2 right

```
struct glthread_* glthread_::right
```

Pointer to the right node in the linked list.

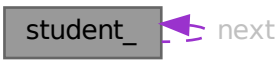
The documentation for this struct was generated from the following file:

- glthread.h

4.5 student_ Struct Reference

Structure representing a student.

Collaboration diagram for student_:



Data Fields

- char name [32]
- uint32_t roll_no
- uint32_t marks_phys
- uint32_t marks_chem
- uint32_t marks_maths
- struct student_ * next

4.5.1 Detailed Description

Structure representing a student.

This structure defines the attributes of a student, including their name, roll number, and subject marks. Additionally, it contains a pointer to the next student in a linked list.

4.5.2 Field Documentation

4.5.2.1 marks_chem

```
uint32_t student_::marks_chem
```

The marks obtained in Chemistry.

4.5.2.2 marks_maths

```
uint32_t student_::marks_maths
```

The marks obtained in Mathematics.

4.5.2.3 marks_phys

```
uint32_t student_::marks_phys
```

The marks obtained in Physics.

4.5.2.4 name

```
char student_::name[32]
```

The name of the student.

4.5.2.5 next

```
struct student_* student_::next
```

Pointer to the next student in the linked list.

4.5.2.6 roll_no

```
uint32_t student_::roll_no
```

The roll number of the student.

The documentation for this struct was generated from the following file:

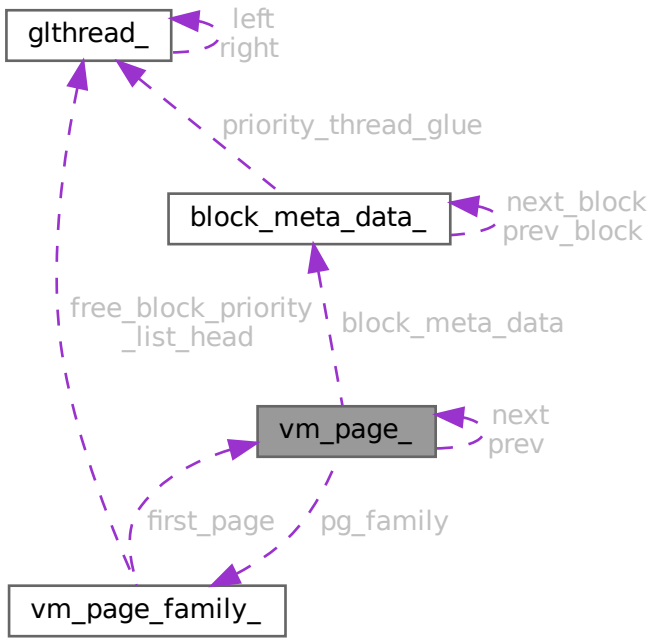
- memory_manager_test.c

4.6 vm_page_ Struct Reference

Structure representing a virtual memory page.

`#include <memory_manager.h>`

Collaboration diagram for vm_page_:



Data Fields

- struct vm_page_ * next
- struct vm_page_ * prev
- struct vm_page_family_ * pg_family
- block_meta_data_t block_meta_data
- char page_memory [0]

4.6.1 Detailed Description

Structure representing a virtual memory page.

This structure represents a virtual memory page used in memory management systems. It contains metadata for managing memory blocks within the page, as well as the actual memory region allocated for storing data blocks.

4.6.2 Field Documentation

4.6.2.1 block_meta_data

```
block_meta_data_t vm_page_::block_meta_data
```

Metadata for managing memory blocks within the page.

4.6.2.2 next

```
struct vm_page_* vm_page_::next
```

Pointer to the next virtual memory page.

4.6.2.3 page_memory

```
char vm_page_::page_memory[0]
```

Memory region allocated for storing data blocks.

4.6.2.4 pg_family

```
struct vm_page_family_* vm_page_::pg_family
```

Pointer to the page family associated with the page.

4.6.2.5 prev

```
struct vm_page_* vm_page_::prev
```

Pointer to the previous virtual memory page.

The documentation for this struct was generated from the following file:

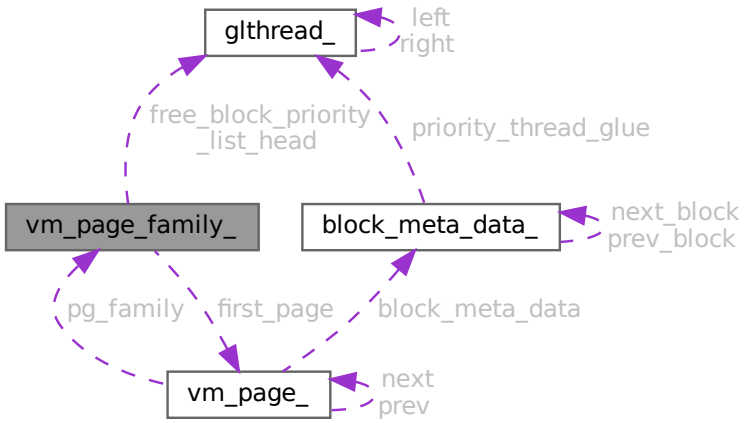
- memory_manager.h

4.7 vm_page_family_ Struct Reference

Structure representing a page family in virtual memory.

`#include <memory_manager.h>`

Collaboration diagram for vm_page_family_:



Data Fields

- char struct_name [MM_MAX_STRUCT_NAME]
- uint32_t struct_size
- vm_page_t * first_page
- glthread_t free_block_priority_list_head

4.7.1 Detailed Description

Structure representing a page family in virtual memory.

This structure maintains information about a page family in virtual memory, including the name of the structure, its size, a pointer to the most recent virtual memory page in use, and a priority list of free memory blocks.

4.7.2 Field Documentation

4.7.2.1 first_page

`vm_page_t* vm_page_family_::first_page`

Pointer to the most recent vm page in use.

4.7.2.2 free_block_priority_list_head

```
glthread_t vm_page_family_::free_block_priority_list_head
```

Priority list of free memory blocks.

4.7.2.3 struct_name

```
char vm_page_family_::struct_name[MM_MAX_STRUCT_NAME]
```

Name of the structure.

4.7.2.4 struct_size

```
uint32_t vm_page_family_::struct_size
```

Size of the structure.

The documentation for this struct was generated from the following file:

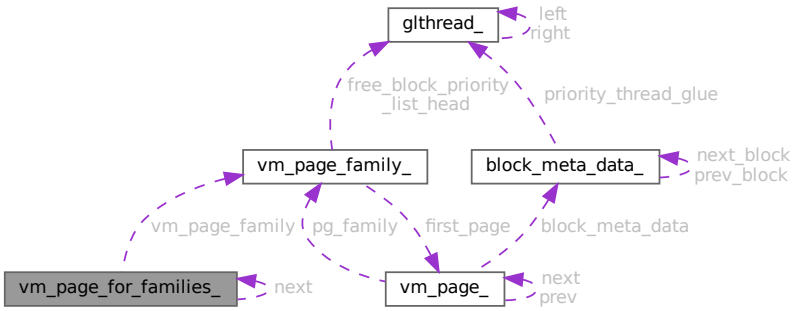
- memory_manager.h

4.8 vm_page_for_families_ Struct Reference

Structure representing a virtual memory page containing families of memory structures.

```
#include <memory_manager.h>
```

Collaboration diagram for vm_page_for_families_:



Data Fields

- struct vm_page_for_families_ * next
- vm_page_family_t vm_page_family [0]

4.8.1 Detailed Description

Structure representing a virtual memory page containing families of memory structures.

4.8.2 Field Documentation

4.8.2.1 next

```
struct vm_page_for_families_* vm_page_for_families_::next
```

Pointer to the next virtual memory page.

4.8.2.2 vm_page_family

```
vm_page_family_t vm_page_for_families_::vm_page_family[0]
```

Array of variable size storing memory structure families.

The documentation for this struct was generated from the following file:

- memory_manager.h

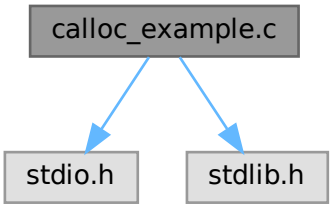
Chapter 5

File Documentation

5.1 calloc_example.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for calloc_example.c:



Functions

- int main ()

5.1.1 Function Documentation

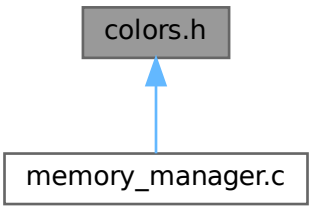
5.1.1.1 main()

```
int main ( )
```

5.2 colors.h File Reference

Header file for ANSI color definitions.

This graph shows which files directly or indirectly include this file:



Macros

- #define ANSI_COLOR_RED "\x1b[31m"
- #define ANSI_COLOR_GREEN "\x1b[32m"
- #define ANSI_COLOR_YELLOW "\x1b[33m"
- #define ANSI_COLOR_BLUE "\x1b[34m"
- #define ANSI_COLOR_MAGENTA "\x1b[35m"
- #define ANSI_COLOR_CYAN "\x1b[36m"
- #define ANSI_COLOR_RESET "\x1b[0m"

5.2.1 Detailed Description

Header file for ANSI color definitions.

This file contains ANSI color definitions used for terminal output coloring.

5.2.2 Macro Definition Documentation

5.2.2.1 ANSI_COLOR_BLUE

```
#define ANSI_COLOR_BLUE "\x1b[34m"
```

Blue ANSI color code.

5.2.2.2 ANSI_COLOR_CYAN

```
#define ANSI_COLOR_CYAN "\x1b[36m"
```

Cyan ANSI color code.

5.2.2.3 ANSI_COLOR_GREEN

```
#define ANSI_COLOR_GREEN "\x1b[32m"
```

Green ANSI color code.

5.2.2.4 ANSI_COLOR_MAGENTA

```
#define ANSI_COLOR_MAGENTA "\x1b[35m"
```

Magenta ANSI color code.

5.2.2.5 ANSI_COLOR_RED

```
#define ANSI_COLOR_RED "\x1b[31m"
```

Red ANSI color code.

5.2.2.6 ANSI_COLOR_RESET

```
#define ANSI_COLOR_RESET "\x1b[0m"
```

ANSI color reset code. COLORS_H_

5.2.2.7 ANSI_COLOR_YELLOW

```
#define ANSI_COLOR_YELLOW "\x1b[33m"
```

Yellow ANSI color code.

5.3 colors.h

Go to the documentation of this file.

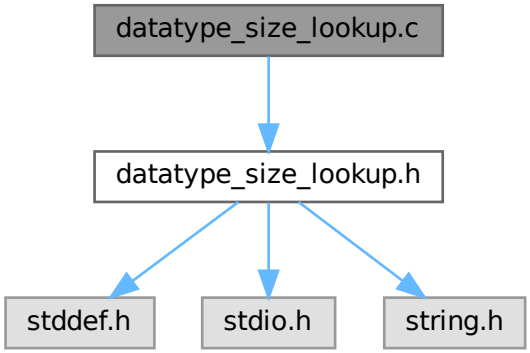
```
00001
00008 #ifndef COLORS_H_
00009 #define COLORS_H_
00010
00011 #define ANSI_COLOR_RED "\x1b[31m"
00012 #define ANSI_COLOR_GREEN "\x1b[32m"
00013 #define ANSI_COLOR_YELLOW "\x1b[33m"
00014 #define ANSI_COLOR_BLUE "\x1b[34m"
00015 #define ANSI_COLOR_MAGENTA "\x1b[35m"
00016 #define ANSI_COLOR_CYAN "\x1b[36m"
00017 #define ANSI_COLOR_RESET "\x1b[0m"
00019 #endif
```

5.4 datatype_size_lookup.c File Reference

Implementation file for data type size lookup functionality.

```
#include "datatype_size_lookup.h"
```

Include dependency graph for datatype_size_lookup.c:



Data Structures

- struct datatype_mapping_t

  Structure representing a data type mapping.

Functions

- size_t get_size_of_datatype (const char *data_type)

  Gets the size of a data type by its name.

Variables

- datatype_mapping_t type_mappings [ ]

  Array of data type mappings.

5.4.1 Detailed Description

Implementation file for data type size lookup functionality.

This file contains the implementation of functions related to retrieving the size of data types.

5.4.2 Function Documentation

5.4.2.1 get_size_of_datatype()

```
size_t get_size_of_datatype (
            const char * data_type )
```

Gets the size of a data type by its name.

This function searches for the given data type name in a pre-defined mapping table and returns the size of the data type if found.

Parameters

data_type	The name of the data type to get the size of.

Returns

The size of the data type if found, otherwise 0. DATATYPE_SIZE_LOOKUP_H_

Here is the caller graph for this function:

5.4.3 Variable Documentation

5.4.3.1 type_mappings

```
datatype_mapping_t type_mappings[]
```

Initial value:

```
= {
    {"int", sizeof(int)},
    {"char", sizeof(char)},
    {"float", sizeof(float)},
    {"double", sizeof(double)},
    {"short", sizeof(short)},
    {"long", sizeof(long)},
    {"long long", sizeof(long long)},
    {"unsigned int", sizeof(unsigned int)},
    {"unsigned char", sizeof(unsigned char)},
    {"unsigned short", sizeof(unsigned short)},
    {"unsigned long", sizeof(unsigned long)},
    {"unsigned long long", sizeof(unsigned long long)},
}
```

Array of data type mappings.

This array contains mappings between data type names and their corresponding sizes.

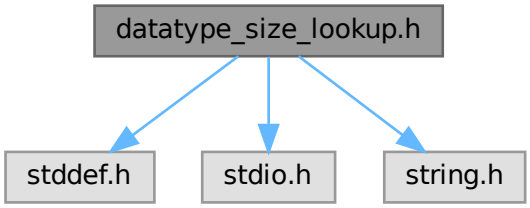
5.5 datatype_size_lookup.h File Reference

Header file for data type size lookup functionality and mappings.

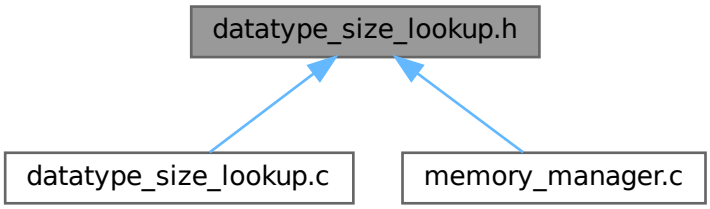
```
#include <stddef.h>
#include <stdio.h>
```

```
#include <string.h>
```

Include dependency graph for datatype_size_lookup.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define MAX_STRUCT_NAME_LEN 50

  Maximum length of a data type name.

Functions

- size_t get_size_of_datatype (const char *data_type)

  Gets the size of a data type by its name.

5.5.1 Detailed Description

Header file for data type size lookup functionality and mappings.

This file contains declarations and mappings for functions and structures related to retrieving the size of data types.

5.5.2 Macro Definition Documentation

5.5.2.1 MAX_STRUCT_NAME_LEN

```
#define MAX_STRUCT_NAME_LEN 50
```

Maximum length of a data type name.

Defines the maximum length allowed for a data type name.

5.5.3 Function Documentation

5.5.3.1 get_size_of_datatype()

```
size_t get_size_of_datatype (
            const char * data_type )
```

Gets the size of a data type by its name.

This function searches for the given data type name in a pre-defined mapping table and returns the size of the data type if found.

Parameters

data_type	The name of the data type to get the size of.

Returns

The size of the data type if found, otherwise 0. DATATYPE_SIZE_LOOKUP_H_

Here is the caller graph for this function:

5.6 datatype_size_lookup.h

Go to the documentation of this file.

```
00001
00009 #ifndef DATATYPE_SIZE_LOOKUP_H_
00010 #define DATATYPE_SIZE_LOOKUP_H_
00011
00012 #include <stddef.h>
00013 #include <stdio.h>
```

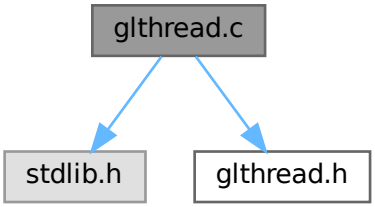
```
00014 #include <string.h>
00015
00022 #define MAX_STRUCT_NAME_LEN 50
00023
00033 size_t get_size_of_datatype(const char *data_type);
00034
00035 #endif
```

5.7 glthread.c File Reference

Implementation file for the Generic Linked List Thread library.

```
#include <stdlib.h>
#include "glthread.h"
```

Include dependency graph for glthread.c:



Functions

- void init_glthread (glthread_t *glthread)

  Initialize a glthread_t structure.
- void glthread_add_next (glthread_t *curr_glthread, glthread_t *new_glthread)

  Adds a new node after the specified current node in the threaded linked list.
- void glthread_add_before (glthread_t *curr_glthread, glthread_t *new_glthread)

  Adds a new node before the specified current node in the threaded linked list.
- void remove_glthread (glthread_t *curr_glthread)

  Removes the specified node from the threaded linked list.
- void delete_glthread_list (glthread_t *base_glthread)

  Delete all glthread_t structures in a linked list.
- void glthread_add_last (glthread_t *base_glthread, glthread_t *new_glthread)

  Add a new glthread_t structure at the last position of a linked list.
- unsigned int get_glthread_list_count (glthread_t *base_glthread)

  Get the count of glthread_t structures in a linked list.
- void glthread_priority_insert (glthread_t *base_glthread, glthread_t *glthread, int(*comp_fn)(void *, void *), int offset)

  Insert a glthread_t structure into a sorted linked list based on priority.
- void * glthread_search (glthread_t *base_glthread, void *(*thread_to_struct_fn)(glthread_t *), void *key, int(*comparison_fn)(void *, void *))

  Search for a specific glthread_t structure in the linked list.

5.7.1 Detailed Description

Implementation file for the Generic Linked List Thread library.

This file contains the implementation of the Generic Linked List Thread library, which provides functionality for managing generic linked lists using threads.

5.7.2 Function Documentation

5.7.2.1 delete_glthread_list()

```
void delete_glthread_list (
            glthread_t * base_glthread )
```

Delete all glthread_t structures in a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).

Here is the call graph for this function:

5.7.2.2 get_glthread_list_count()

```
unsigned int get_glthread_list_count (
            glthread_t * base_glthread )
```

Get the count of glthread_t structures in a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).

Returns

unsigned int The count of glthread_t structures in the linked list.

5.7.2.3 glthread_add_before()

```
void glthread_add_before (
            glthread_t * base_glthread,
            glthread_t * new_glthread )
```

Adds a new node before the specified current node in the threaded linked list.

This function adds a new node before the specified current node in the threaded linked list.

Parameters

curr_glthread	Pointer to the current node in the threaded linked list.
new_glthread	Pointer to the new node to be added.

5.7.2.4 glthread_add_last()

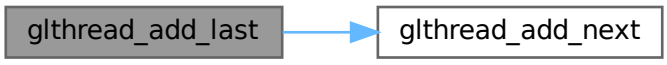
```
void glthread_add_last (
            glthread_t * base_glthread,
            glthread_t * new_glthread )
```

Add a new glthread_t structure at the last position of a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
new_glthread	Pointer to the new glthread_t structure to be added.

Here is the call graph for this function:



5.7.2.5 glthread_add_next()

```
void glthread_add_next (
            glthread_t * base_glthread,
            glthread_t * new_glthread )
```

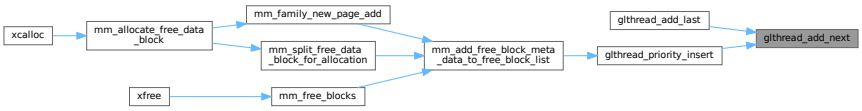
Adds a new node after the specified current node in the threaded linked list.

This function adds a new node after the specified current node in the threaded linked list.

Parameters

curr_glthread	Pointer to the current node in the threaded linked list.
new_glthread	Pointer to the new node to be added.

Here is the caller graph for this function:



5.7.2.6 glthread_priority_insert()

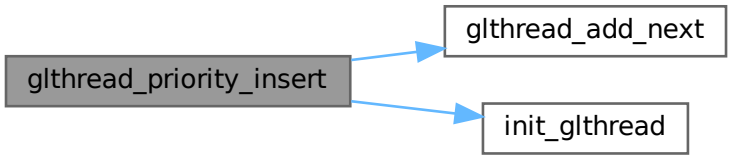
```
void glthread_priority_insert (
            glthread_t * base_glthread,
            glthread_t * glthread,
            int(*)(void *, void *) comp_fn,
            int offset )
```

Insert a glthread_t structure into a sorted linked list based on priority.

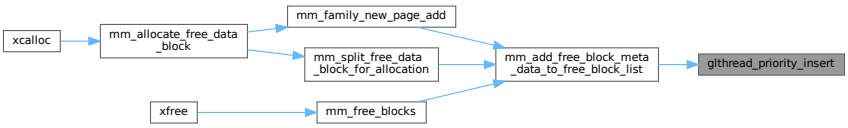
Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
glthread	Pointer to the glthread_t structure to be inserted.
comp_fn	Pointer to the comparison function used to determine priority.
offset	Offset to access the user data within the glthread_t structure.

< Add in the endHere is the call graph for this function:



Here is the caller graph for this function:



5.7.2.7 glthread_search()

```
void * glthread_search (
            glthread_t * base_glthread,
            void *(*)(glthread_t *) thread_to_struct_fn,
            void * key,
            int(*)(void *, void *) comparison_fn )
```

Search for a specific glthread_t structure in the linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
thread_to_struct↩ _fn	Function pointer to convert a glthread_t structure to the corresponding user-defined structure.
key	Pointer to the key used for comparison.
comparison_fn	Function pointer to the comparison function used to compare keys.

Returns

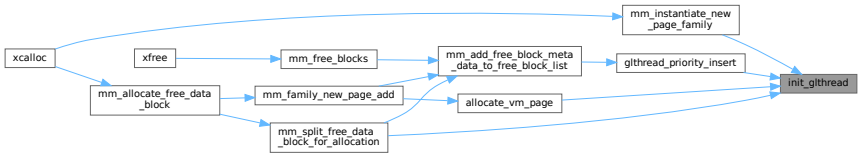
void∗ Pointer to the user-defined structure corresponding to the found glthread_t structure, or NULL if not found.

5.7.2.8 init_glthread()

```
void init_glthread (
            glthread_t * glthread )
```

Initialize a glthread_t structure.

< System includes < Project includes Here is the caller graph for this function:



5.7.2.9 remove_glthread()

```
void remove_glthread (
            glthread_t * glthread )
```

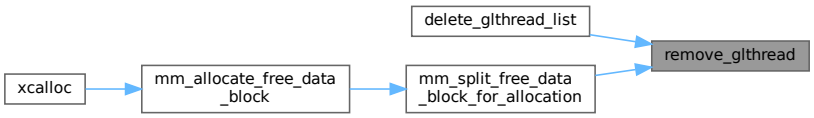
Removes the specified node from the threaded linked list.

This function removes the specified node from the threaded linked list.

Parameters

curr_glthread	Pointer to the node to be removed.

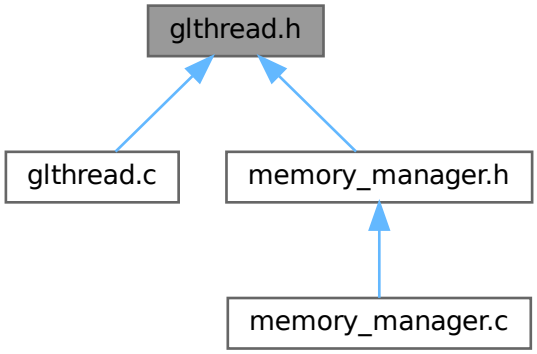
Here is the caller graph for this function:



5.8 glthread.h File Reference

Header file for the Generic Linked List Thread library.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct glthread_

  Structure representing a generic linked list node for threaded linking.

Macros

- #define IS_GLTHREAD_LIST_EMPTY(glthreadptr) ((glthreadptr)->right == 0 && (glthreadptr)->left == 0)

  Macro to check if a linked list of glthread_t structures is empty.
- #define GLTHREAD_TO_STRUCT(fn_name, structure_name, field_name, glthreadptr)

  Macro to convert a glthread_t structure to a user-defined structure.
- #define BASE(glthreadptr) ((glthreadptr)->right)

  Macro to retrieve the base of a linked list.
- #define ITERATE_GLTHREAD_BEGIN(glthreadptrstart, glthreadptr)

  Macro to begin iterating over a linked list of glthread_t structures.
- #define ITERATE_GLTHREAD_END(glthreadptrstart, glthreadptr)

  Macro to end iteration over a linked list of glthread_t structures.
- #define GLTHREAD_GET_USER_DATA_FROM_OFFSET(glthreadptr, offset) (void *)((char *)(glthreadptr)-offset)

  Macro to get a pointer to user-defined data from a glthread_t pointer and offset.

Typedefs

- typedef struct glthread_ glthread_t

  Structure representing a generic linked list node for threaded linking.

Functions

- void init_glthread (glthread_t *glthread)

  Initialize a glthread_t structure.
- void glthread_add_next (glthread_t *base_glthread, glthread_t *new_glthread)

  Adds a new node after the specified current node in the threaded linked list.
- void glthread_add_before (glthread_t *base_glthread, glthread_t *new_glthread)

  Adds a new node before the specified current node in the threaded linked list.
- void remove_glthread (glthread_t *glthread)

  Removes the specified node from the threaded linked list.
- void glthread_add_last (glthread_t *base_glthread, glthread_t *new_glthread)

  Add a new glthread_t structure at the last position of a linked list.
- void delete_glthread_list (glthread_t *base_glthread)

  Delete all glthread_t structures in a linked list.
- unsigned int get_glthread_list_count (glthread_t *base_glthread)

  Get the count of glthread_t structures in a linked list.
- void glthread_priority_insert (glthread_t *base_glthread, glthread_t *glthread, int(*comp_fn)(void *, void *), int offset)

  Insert a glthread_t structure into a sorted linked list based on priority.
- void * glthread_search (glthread_t *base_glthread, void *(*thread_to_struct_fn)(glthread_t *), void *key, int(*comparison_fn)(void *, void *))

  Search for a specific glthread_t structure in the linked list.

5.8.1 Detailed Description

Header file for the Generic Linked List Thread library.

This header file contains the interface for the Generic Linked List Thread library, which provides functionality for managing generic linked lists using threaded linking.

5.8.2 Macro Definition Documentation

5.8.2.1 BASE

```
#define BASE(
              glthreadptr ) ((glthreadptr)->right)
```

Macro to retrieve the base of a linked list.

This macro retrieves the base of a linked list given a pointer to a glthread_t structure. It returns the pointer to the right child of the provided glthread_t structure, which is typically the base of the linked list.

Parameters

glthreadptr	Pointer to the glthread_t structure from which to retrieve the base.

Returns

glthread_t* Pointer to the base of the linked list.

5.8.2.2 GLTHREAD_GET_USER_DATA_FROM_OFFSET

```
#define GLTHREAD_GET_USER_DATA_FROM_OFFSET(
              glthreadptr,
              offset )   (void *)((char *)(glthreadptr)-offset)
```

Macro to get a pointer to user-defined data from a glthread_t pointer and offset.

This macro calculates a pointer to user-defined data from a given glthread_t pointer and an offset. It's useful when you have a glthread_t pointer embedded within a user-defined structure and you want to access the user-defined data based on a known offset.

Parameters

glthreadptr	Pointer to the glthread_t structure.
offset	Offset of the glthread_t member within the user-defined structure.

Returns

Pointer to the user-defined data. GLUETHREAD_H_

5.8.2.3 GLTHREAD_TO_STRUCT

```
#define GLTHREAD_TO_STRUCT(
              fn_name,
              structure_name,
              field_name,
              glthreadptr )
```

Value:

```
static inline structure_name *fn_name(glthread_t *glthreadptr) {            \
  return (structure_name *)((char *)(glthreadptr) -                         \
                            (char *)&(((structure_name *)0)->field_name));  \
}
```

Macro to convert a glthread_t structure to a user-defined structure.

This macro provides a convenient way to convert a glthread_t structure to a user-defined structure. It creates a conversion function that takes a glthread_t pointer and returns a pointer to the user-defined structure. This is particularly useful when you have a glthread_t structure embedded within a user-defined structure and need to access the user-defined data.

Parameters

fn_name	The name of the conversion function to be created.
struct_type	The type of the user-defined structure.
glthread_member	The name of the glthread_t member within the user-defined structure.
glthread_ptr	The name of the glthread_t pointer variable.

Example usage: Suppose we have a user-defined structure named `block_meta_data_t` that contains a `glthread_t` member named `priority_thread_glue`. We want to create a conversion function named `glthread_to_block_meta_data` to convert a `glthread_t` pointer to a `block_meta_data_t` pointer:

GLTHREAD_TO_STRUCT(glthread_to_block_meta_data, block_meta_data_t, priority_thread_glue, glthread_ptr);

Now, we can use `glthread_to_block_meta_data` to convert `glthread_t` pointers to `block_meta_data_t` pointers and access the metadata associated with memory blocks.

5.8.2.4 IS_GLTHREAD_LIST_EMPTY

```
#define IS_GLTHREAD_LIST_EMPTY(
              glthreadptr )   ((glthreadptr)->right == 0 && (glthreadptr)->left == 0)
```

Macro to check if a linked list of glthread_t structures is empty.

This macro checks if a linked list of glthread_t structures is empty. It evaluates to true if both the right and left pointers of the provided glthread_t structure are NULL, indicating an empty list.

Parameters

glthreadptr	Pointer to the glthread_t structure representing the linked list.

Returns

int Returns 1 if the linked list is empty, 0 otherwise.

5.8.2.5 ITERATE_GLTHREAD_BEGIN

```
#define ITERATE_GLTHREAD_BEGIN(
              glthreadptrstart,
              glthreadptr )
```

Value:

```
{                                                                       \
  glthread_t *_glthread_ptr = NULL;                                     \
  glthreadptr = BASE(glthreadptrstart);                                 \
  for (; glthreadptr != NULL; glthreadptr = _glthread_ptr) {            \
    _glthread_ptr = (glthreadptr)->right;
```

Macro to begin iterating over a linked list of glthread_t structures.

This macro sets up a loop to iterate over a linked list of glthread_t structures. It initializes necessary variables and pointers for the iteration.

Parameters

glthreadptrstart	Pointer to the starting glthread_t structure for iteration.
glthreadptr	Pointer to the current glthread_t structure being iterated.

5.8.2.6 ITERATE_GLTHREAD_END

```
#define ITERATE_GLTHREAD_END(
              glthreadptrstart,
              glthreadptr )
```

Value:

```
  }                                                                     \
  }
```

Macro to end iteration over a linked list of glthread_t structures.

This macro marks the end of the loop initiated by ITERATE_GLTHREAD_BEGIN. It closes the loop block.

Parameters

glthreadptrstart	Pointer to the starting glthread_t structure for iteration.
glthreadptr	Pointer to the current glthread_t structure being iterated.

5.8.3 Typedef Documentation

5.8.3.1 glthread_t

```
typedef struct glthread_ glthread_t
```

Structure representing a generic linked list node for threaded linking.

This structure defines a generic linked list node for threaded linking. It consists of left and right pointers for threading the nodes together.

5.8.4 Function Documentation

5.8.4.1 delete_glthread_list()

```
void delete_glthread_list (
            glthread_t * base_glthread )
```

Delete all glthread_t structures in a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).

Here is the call graph for this function:

5.8.4.2 get_glthread_list_count()

```
unsigned int get_glthread_list_count (
             glthread_t * base_glthread )
```

Get the count of glthread_t structures in a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).

Returns

unsigned int The count of glthread_t structures in the linked list.

5.8.4.3 glthread_add_before()

```
void glthread_add_before (
             glthread_t * base_glthread,
             glthread_t * new_glthread )
```

Adds a new node before the specified current node in the threaded linked list.

This function adds a new node before the specified current node in the threaded linked list.

Parameters

curr_glthread	Pointer to the current node in the threaded linked list.
new_glthread	Pointer to the new node to be added.

5.8.4.4 glthread_add_last()

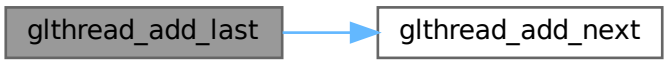
```
void glthread_add_last (
            glthread_t * base_glthread,
            glthread_t * new_glthread )
```

Add a new glthread_t structure at the last position of a linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
new_glthread	Pointer to the new glthread_t structure to be added.

Here is the call graph for this function:



5.8.4.5 glthread_add_next()

```
void glthread_add_next (
            glthread_t * base_glthread,
            glthread_t * new_glthread )
```

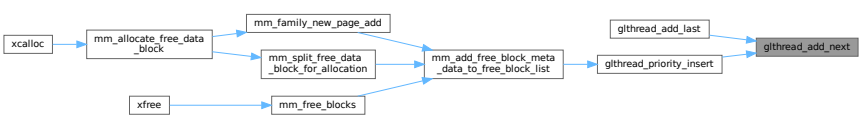
Adds a new node after the specified current node in the threaded linked list.

This function adds a new node after the specified current node in the threaded linked list.

Parameters

curr_glthread	Pointer to the current node in the threaded linked list.
new_glthread	Pointer to the new node to be added.

Here is the caller graph for this function:



5.8.4.6 glthread_priority_insert()

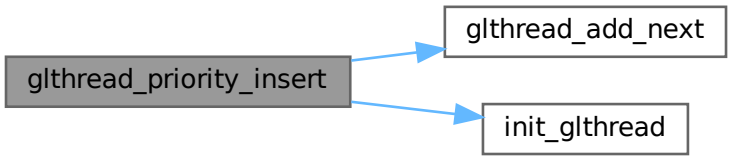
```
void glthread_priority_insert (
            glthread_t * base_glthread,
            glthread_t * glthread,
            int(*)(void *, void *) comp_fn,
            int offset )
```

Insert a glthread_t structure into a sorted linked list based on priority.

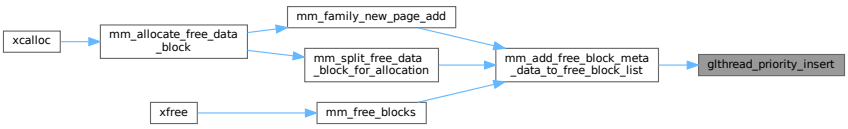
Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
glthread	Pointer to the glthread_t structure to be inserted.
comp_fn	Pointer to the comparison function used to determine priority.
offset	Offset to access the user data within the glthread_t structure.

< Add in the endHere is the call graph for this function:



Here is the caller graph for this function:



5.8.4.7 glthread_search()

```
void * glthread_search (
            glthread_t * base_glthread,
            void *(*)(glthread_t *) thread_to_struct_fn,
            void * key,
            int(*)(void *, void *) comparison_fn )
```

Search for a specific glthread_t structure in the linked list.

Parameters

base_glthread	Pointer to the base glthread_t structure (head of the linked list).
thread_to_struct↩ _fn	Function pointer to convert a glthread_t structure to the corresponding user-defined structure.
key	Pointer to the key used for comparison.
comparison_fn	Function pointer to the comparison function used to compare keys.

Returns

void∗ Pointer to the user-defined structure corresponding to the found glthread_t structure, or NULL if not found.

5.8.4.8 init_glthread()

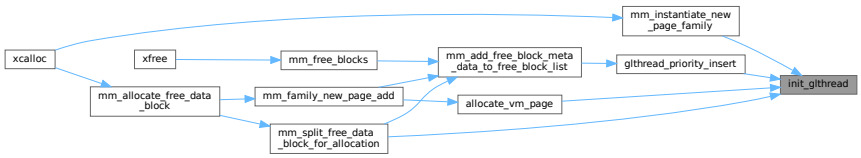
```
void init_glthread (
            glthread_t * glthread )
```

Initialize a glthread_t structure.

Parameters

glthread	Pointer to the glthread_t structure to be initialized.

< System includes < Project includes Here is the caller graph for this function:



5.8.4.9 remove_glthread()

```
void remove_glthread (
            glthread_t * glthread )
```

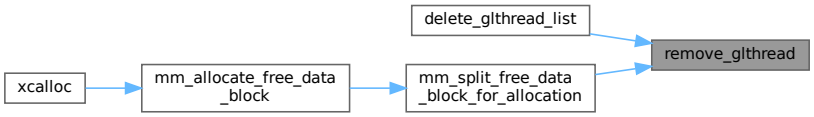
Removes the specified node from the threaded linked list.

This function removes the specified node from the threaded linked list.

Parameters

curr_glthread	Pointer to the node to be removed.

Here is the caller graph for this function:



5.9 glthread.h

Go to the documentation of this file.

```
00001 /*****************************************************************/
00002 /******* Author     : Mahmoud Abdelraouf Mahmoud    ***************/
00003 /******* Date       : 13 Apr 2023                   ***************/
00004 /******* Version    : 0.1                           ***************/
00005 /******* File Name : Generic Linked List Thread.h  ***************/
00006 /*****************************************************************/
00007
00017 #ifndef GLUETHREAD_H_
00018 #define GLUETHREAD_H_
00019
00020 //-----------------< user defined data type section -----------------/
00028 typedef struct glthread_ {
00029   struct glthread_ *left;
00030   struct glthread_ *right;
00031 } glthread_t;
00032
00033 //-----------------< Public functions interface -----------------/
00039 void init_glthread(glthread_t *glthread);
00040
00051 void glthread_add_next(glthread_t *base_glthread, glthread_t *new_glthread);
00052
00063 void glthread_add_before(glthread_t *base_glthread, glthread_t *new_glthread);
00064
00072 void remove_glthread(glthread_t *glthread);
00073
00081 void glthread_add_last(glthread_t *base_glthread, glthread_t *new_glthread);
00082
00089 void delete_glthread_list(glthread_t *base_glthread);
00090
00098 unsigned int get_glthread_list_count(glthread_t *base_glthread);
00099
00110 void glthread_priority_insert(glthread_t *base_glthread, glthread_t *glthread,
00111                               int (*comp_fn)(void *, void *), int offset);
00112
00126 void *glthread_search(glthread_t *base_glthread,
00127                      void *(*thread_to_struct_fn)(glthread_t *), void *key,
00128                      int (*comparison_fn)(void *, void *));
00129 //-----------------< Function-like macro section -----------------/
00141 #define IS_GLTHREAD_LIST_EMPTY(glthreadptr)                                \
00142   ((glthreadptr)->right == 0 && (glthreadptr)->left == 0)
00143
00172 #define GLTHREAD_TO_STRUCT(fn_name, structure_name, field_name, glthreadptr)   \
00173   static inline structure_name *fn_name(glthread_t *glthreadptr) {             \
00174     return (structure_name *)((char *)(glthreadptr) -                          \
00175                               (char *)&(((structure_name *)0)->field_name));   \
00176   }
00177
00190 #define BASE(glthreadptr) ((glthreadptr)->right)
00191
00204 #define ITERATE_GLTHREAD_BEGIN(glthreadptrstart, glthreadptr)                  \
00205   {                                                                            \
00206     glthread_t *_glthread_ptr = NULL;                                          \
00207     glthreadptr = BASE(glthreadptrstart);                                      \
00208     for (; glthreadptr != NULL; glthreadptr = _glthread_ptr) {                 \
00209       _glthread_ptr = (glthreadptr)->right;
00210
00222 #define ITERATE_GLTHREAD_END(glthreadptrstart, glthreadptr)                    \
00223   }                                                                            \
00224   }
00225
00240 #define GLTHREAD_GET_USER_DATA_FROM_OFFSET(glthreadptr, offset)                \
00241   (void *)((char *)(glthreadptr)-offset)
00242
00243 #endif
```

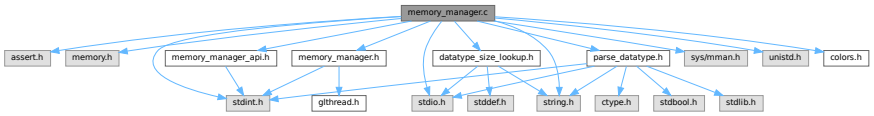
5.10 gperf.md File Reference

5.11 memory_manager.c File Reference

Implementation file for the Memory Manager module.

```
#include <assert.h>
#include <memory.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include "colors.h"
#include "datatype_size_lookup.h"
#include "memory_manager.h"
#include "memory_manager_api.h"
#include "parse_datatype.h"
```

Include dependency graph for memory_manager.c:



Functions

- void mm_init ()

  Initializes the memory manager.
- static void * mm_get_new_vm_page_from_kernel (int units)
- static void mm_return_vm_page_to_kernel (void *vm_page, int units)
- void mm_instantiate_new_page_family (char *struct_name, uint32_t struct_size)

  Instantiates a new page family for a memory structure.
- vm_page_family_t * lookup_page_family_by_name (char *struct_name)

  Looks up a page family by its name.
- static vm_page_t * mm_family_new_page_add (vm_page_family_t *vm_page_family)
- vm_bool_t mm_is_vm_page_empty (vm_page_t *vm_page)

  Checks if a virtual memory page is empty.
- static uint32_t mm_max_page_allocatable_memory (int units)
- vm_page_t * allocate_vm_page (vm_page_family_t *vm_page_family)

  Allocates a new virtual memory page for a given page family.
- void mm_vm_page_delete_and_free (vm_page_t *vm_page)

  Deletes and frees a virtual memory page.
- static void mm_union_free_blocks (block_meta_data_t *first, block_meta_data_t *second)
- static int free_blocks_comparison_function (void *_block_meta_data1, void *_block_meta_data2)
- static void mm_add_free_block_meta_data_to_free_block_list (vm_page_family_t *vm_page_family, block_meta_data_t *free_block)
- static int mm_get_hard_internal_memory_frag_size (block_meta_data_t *first, block_meta_data_t *second)
- static block_meta_data_t * mm_free_blocks (block_meta_data_t *to_be_free_block)
- void xfree (void *app_data)

Frees memory allocated by the memory manager.

- static block_meta_data_t * mm_allocate_free_data_block (vm_page_family_t *vm_page_family, uint32_t req_size)
- static block_meta_data_t * mm_get_biggest_free_block_page_family (vm_page_family_t *vm_page_family)
- static vm_bool_t mm_split_free_data_block_for_allocation (vm_page_family_t *vm_page_family, block_meta_data_t *block_meta_data, uint32_t size)
- void * xcalloc (char *struct_name, int units)

  Allocates and initializes memory for an array of structures.
- void mm_print_registered_page_families ()

  Prints all registered page families.
- void mm_print_block_usage ()

  Prints information about the memory block usage.
- void mm_print_vm_page_details (vm_page_t *vm_page)

  Prints details of a virtual memory page.
- void mm_print_memory_usage (char *struct_name)

  Prints memory usage details related to the memory manager.

Variables

- static size_t SYSTEM_PAGE_SIZE = 0

  Size of the system page.
- static vm_page_for_families_t * first_vm_page_for_families = NULL

  Pointer to the first virtual memory page for page families.

5.11.1 Detailed Description

Implementation file for the Memory Manager module.

This file contains the implementation of functions declared in MemoryManager.h. The Memory Manager module is responsible for managing memory allocation and deallocation, including virtual memory page management, block metadata handling, and allocation algorithms.

5.11.2 Function Documentation

5.11.2.1 allocate_vm_page()

```
vm_page_t * allocate_vm_page (
            vm_page_family_t * vm_page_family )
```

Allocates a new virtual memory page for a given page family.

This function allocates a new virtual memory page for the specified page family. It initializes the metadata and pointers associated with the page and inserts the page into the linked list of pages belonging to the page family.

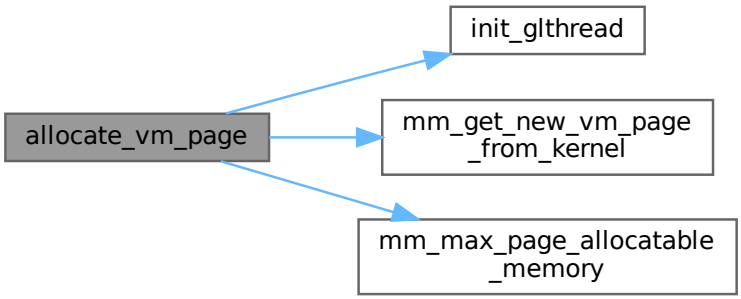
Parameters

vm_page_family	Pointer to the page family for which the page is being allocated.

Returns

Pointer to the newly allocated virtual memory page.

Here is the call graph for this function:

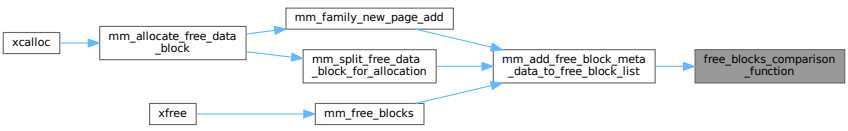


Here is the caller graph for this function:

5.11.2.2 free_blocks_comparison_function()

```
static int free_blocks_comparison_function (
            void * _block_meta_data1,
            void * _block_meta_data2 )  [static]
```

Here is the caller graph for this function:



5.11.2.3 lookup_page_family_by_name()

```
vm_page_family_t * lookup_page_family_by_name (
            char * struct_name )
```

Looks up a page family by its name.

This function iterates over all virtual memory pages hosting page families and returns a pointer to the page family object identified by the given struct_name. If no such page family object is found, it returns NULL.

Parameters

struct_name	The name of the page family to look up.

Returns

Pointer to the page family object if found, otherwise NULL.

Note

This function should be used to retrieve a page family object by its name after the page families have been registered and initialized using the appropriate functions and macros provided by the memory manager.

See also

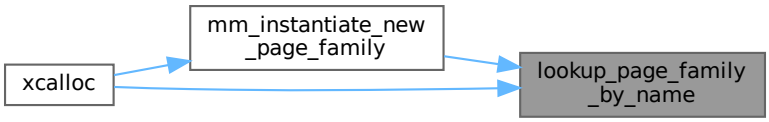
mm_init

MM_REG_STRUCT

vm_page_for_families_t

vm_page_family_t

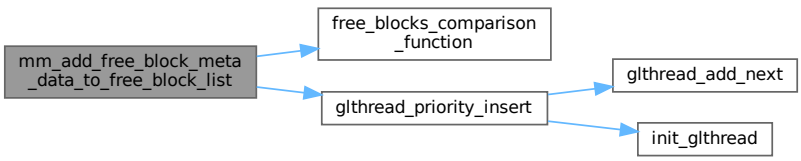
Here is the caller graph for this function:



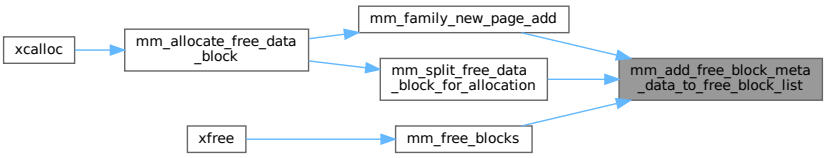
5.11.2.4 mm_add_free_block_meta_data_to_free_block_list()

```
static void mm_add_free_block_meta_data_to_free_block_list (
            vm_page_family_t * vm_page_family,
            block_meta_data_t * free_block )  [static]
```

Here is the call graph for this function:



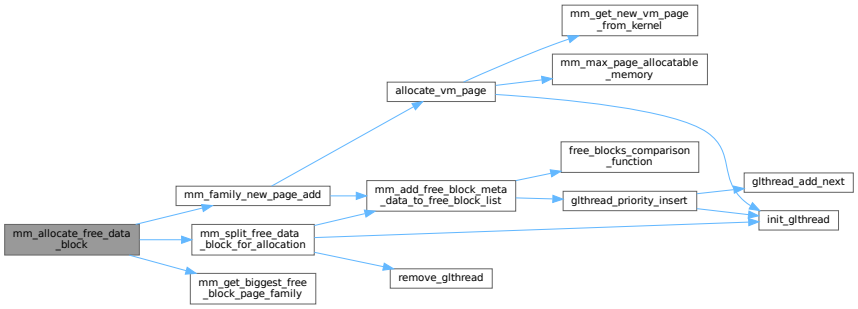
Here is the caller graph for this function:



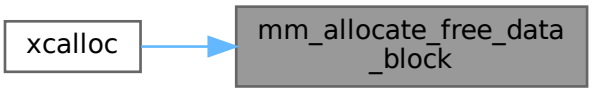
5.11.2.5 mm_allocate_free_data_block()

```
static block_meta_data_t * mm_allocate_free_data_block (
            vm_page_family_t * vm_page_family,
            uint32_t req_size )  [static]
```

Here is the call graph for this function:



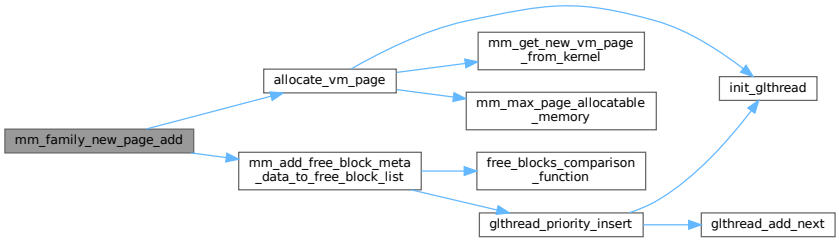
Here is the caller graph for this function:



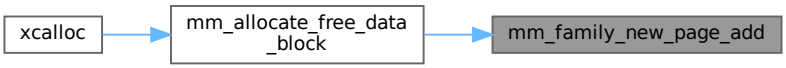
5.11.2.6 mm_family_new_page_add()

```
static vm_page_t * mm_family_new_page_add (
            vm_page_family_t * vm_page_family )  [static]
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.7 mm_free_blocks()

```
static block_meta_data_t * mm_free_blocks (
            block_meta_data_t * to_be_free_block )  [static]
```

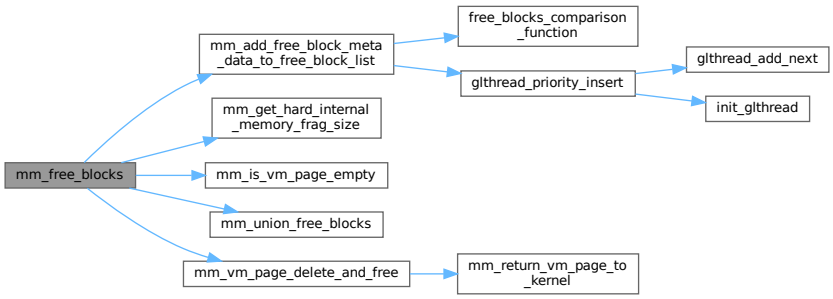
1 Pointer to the freed block

2 Marking the block as free

3 Next block pointer

4 Union two free blocks

5 Delete and free the hosting pageHere is the call graph for this function:

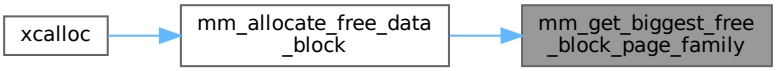


Here is the caller graph for this function:

5.11.2.8 mm_get_biggest_free_block_page_family()

```
static block_meta_data_t * mm_get_biggest_free_block_page_family (
            vm_page_family_t * vm_page_family )  [inline], [static]
```

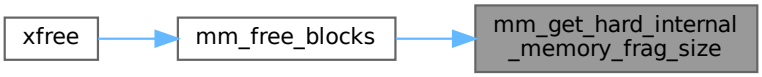
Here is the caller graph for this function:



5.11.2.9 mm_get_hard_internal_memory_frag_size()

```
static int mm_get_hard_internal_memory_frag_size (
            block_meta_data_t * first,
            block_meta_data_t * second )  [static]
```

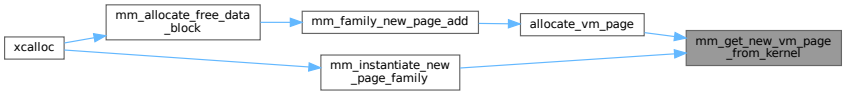
Here is the caller graph for this function:



5.11.2.10 mm_get_new_vm_page_from_kernel()

```
static void * mm_get_new_vm_page_from_kernel (
            int units )  [static]
```

Here is the caller graph for this function:



5.11.2.11 mm_init()

```
void mm_init ( )
```

Initializes the memory manager.

This function initializes the memory manager. It sets up necessary configurations and parameters for memory management operations within the program. It specifically determines the system page size using the `getpagesize()` system call and assigns it to the global variable `SYSTEM_PAGE_SIZE`.

Note

This function should be called before any memory management operations are performed within the program. It is typically called at the beginning of the program execution to ensure proper initialization of memory management functionalities.

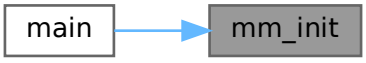
Warning

This function relies on the `getpagesize()` system call to determine the system page size. Therefore, it may not be portable across all platforms. It is primarily intended for use in Unix-like systems where `getpagesize()` is available.

See also

man getpagesize()

Here is the caller graph for this function:



5.11.2.12 mm_instantiate_new_page_family()

```
void mm_instantiate_new_page_family (
            char * struct_name,
            uint32_t struct_size )
```

Instantiates a new page family for a memory structure.

This function creates a new page family for a memory structure identified by its name and size. It allocates memory for the page family and adds it to the existing virtual memory pages if necessary. Each page family can contain multiple memory structures of the same type.

Parameters

struct_name	The name of the memory structure.
struct_size	The size of the memory structure.

Note

If the size of the memory structure exceeds the system page size, an error message is printed, and the function returns without creating the page family.

This function maintains a linked list of virtual memory pages (`first_vm_page_for_families`) to store the page families. If there are no existing pages, it allocates a new page and initializes it with the first page family. If the existing pages are full, it allocates a new page and adds it to the beginning of the linked list.

If a page family with the same name already exists, an assertion error is triggered, indicating a conflict in page family instantiation.

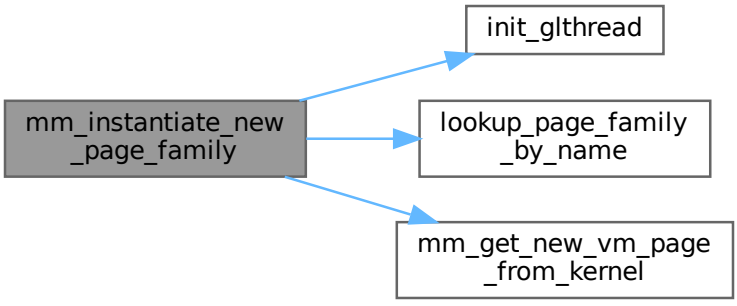
Warning

This function relies on the `mm_get_new_vm_page_from_kernel()` function to allocate memory from the kernel for the page family. Improper use or misuse of this function can lead to memory leaks or system instability.

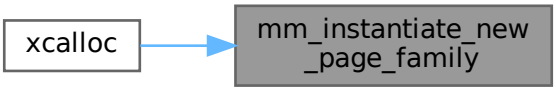
See also

mm_get_new_vm_page_from_kernel()

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.13 mm_is_vm_page_empty()

```
vm_bool_t mm_is_vm_page_empty (
            vm_page_t * vm_page )
```

Checks if a virtual memory page is empty.

This function determines whether a virtual memory page is empty based on its metadata.

Parameters

vm_page	Pointer to the virtual memory page to be checked.

Returns

- MM_TRUE if the page is empty.
- MM_FALSE if the page is not empty or if the input pointer is NULL.

Note

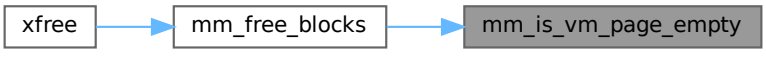
A virtual memory page is considered empty if all the following conditions are met:

- The 'next_block' pointer in the block metadata is NULL, indicating no next block.
- The 'prev_block' pointer in the block metadata is NULL, indicating no previous block.
- The 'is_free' flag in the block metadata is set to MM_TRUE, indicating the page is free.

Warning

It is important to ensure that the 'vm_page' parameter is a valid pointer to a virtual memory page structure. Passing invalid or uninitialized pointers may result in undefined behavior.

Here is the caller graph for this function:



5.11.2.14 mm_max_page_allocatable_memory()

```
static uint32_t mm_max_page_allocatable_memory (
            int units )  [inline], [static]
```

Here is the caller graph for this function:

5.11.2.15 mm_print_block_usage()

```
void mm_print_block_usage ( )
```

Prints information about the memory block usage.

This function iterates through all virtual memory pages and their associated memory block families to print information about the memory block usage, including the total block count, free block count, occupied block count, and application memory usage. Here is the caller graph for this function:

5.11.2.16 mm_print_memory_usage()

```
void mm_print_memory_usage (
            char * struct_name )
```

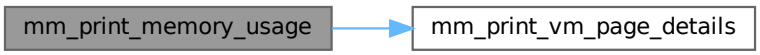
Prints memory usage details related to the memory manager.

This function prints information about the memory usage of the memory manager, including details of each virtual memory page family and the total memory being used. Optionally, it can filter the output by a specific structure name.

Parameters

struct_name	Optional parameter to filter the output by a specific structure name.

Here is the call graph for this function:



Here is the caller graph for this function:

5.11.2.17 mm_print_registered_page_families()

```
void mm_print_registered_page_families ( )
```

Prints all registered page families.

This function prints all page families that have been registered with the Linux Memory Manager. It iterates over all virtual memory pages hosting page families and prints information about each page family, including its name and size.

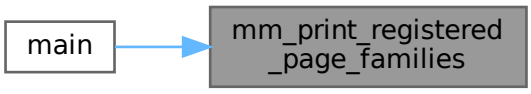
Note

This function should be invoked after the application has performed registration for all its structures using the `MM_REG_STRUCT` macro. It relies on the `first_vm_page_for_families` global variable, which maintains a linked list of virtual memory pages containing page families.

See also

MM_REG_STRUCT

Here is the caller graph for this function:



5.11.2.18 mm_print_vm_page_details()

```
void mm_print_vm_page_details (
            vm_page_t * vm_page )
```

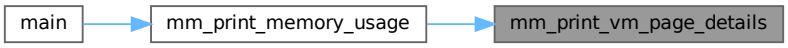
Prints details of a virtual memory page.

This function prints detailed information about a virtual memory page, including its next and previous pointers, page family name, and information about each block within the page.

Parameters

vm_page	Pointer to the virtual memory page.

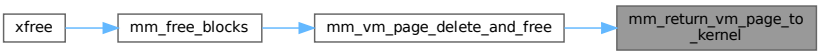
Here is the caller graph for this function:



5.11.2.19 mm_return_vm_page_to_kernel()

```
static void mm_return_vm_page_to_kernel (
            void * vm_page,
            int units )  [static]
```

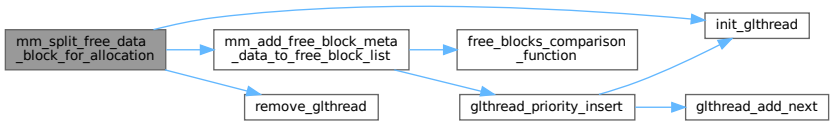
Here is the caller graph for this function:



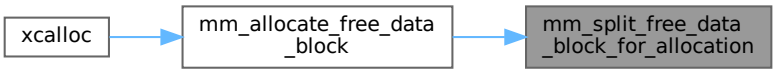
5.11.2.20 mm_split_free_data_block_for_allocation()

```
static vm_bool_t mm_split_free_data_block_for_allocation (
            vm_page_family_t * vm_page_family,
            block_meta_data_t * block_meta_data,
            uint32_t size )  [static]
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.21 mm_union_free_blocks()

```
static void mm_union_free_blocks (
            block_meta_data_t * first,
            block_meta_data_t * second )  [static]
```

Here is the caller graph for this function:

5.11.2.22 mm_vm_page_delete_and_free()

```
void mm_vm_page_delete_and_free (
            vm_page_t * vm_page )
```

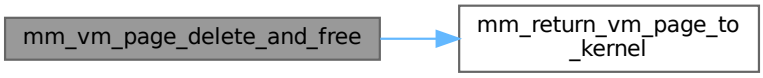
Deletes and frees a virtual memory page.

This function deletes and frees a virtual memory page. It removes the page from the linked list of pages belonging to its page family and deallocates the memory associated with the page.

Parameters

vm_page	Pointer to the virtual memory page to be deleted and freed.

Here is the call graph for this function:



Here is the caller graph for this function:

5.11.2.23 xcalloc()

```
void * xcalloc (
            char * struct_name,
            int units )
```

Allocates and initializes memory for an array of structures.

This function allocates memory for an array of structures of the specified type and initializes the memory to zero. It first looks up the page family associated with the specified structure name to determine the size of the structure. Then, it checks if the requested memory size exceeds the maximum allocatable memory per page. If the allocation is successful, it initializes the allocated memory to zero and returns a pointer to the allocated memory.

Parameters

struct_name	The name of the structure type for which memory is to be allocated.
units	The number of structures to allocate.

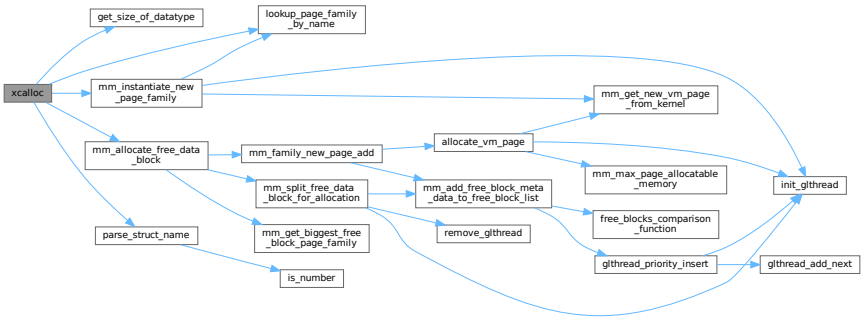
Returns

A pointer to the allocated memory if successful, or NULL if the allocation fails.

Note

This function assumes that the specified structure type has been registered with the Memory Manager using the mm_register_structure function. It also assumes that the specified structure type has a corresponding page family registered with the Memory Manager.

Here is the call graph for this function:



5.11.2.24 xfree()

```
void xfree (
            void * app_data )
```

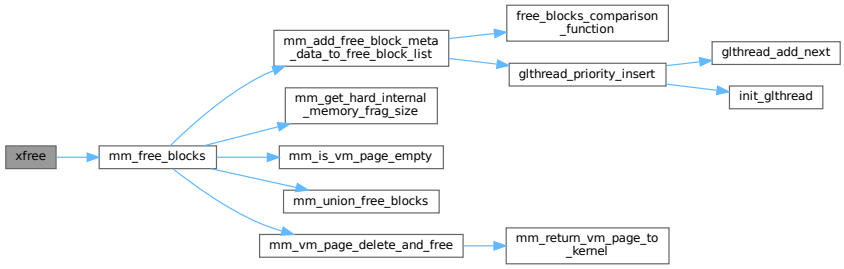
Frees memory allocated by the memory manager.

This function is used to free memory that was previously allocated by the memory manager. It takes a pointer to the memory to be freed as its argument. The pointer is adjusted to point to the block metadata, and then it is passed to the memory manager's free blocks function.

Parameters

app_data	Pointer to the memory to be freed.

Here is the call graph for this function:



5.11.3 Variable Documentation

5.11.3.1 first_vm_page_for_families

```
vm_page_for_families_t* first_vm_page_for_families = NULL  [static]
```

Pointer to the first virtual memory page for page families.

This static pointer variable holds the address of the first virtual memory page used to store page families. It is initialized to NULL, indicating that no virtual memory page is currently allocated for page families. As page families are instantiated, new virtual memory pages may be allocated and linked to this pointer.

Note

This variable is static and should be accessible only within the scope of the file in which it is declared. It is used to maintain the linked list of virtual memory pages for page families throughout the program.

5.11.3.2 SYSTEM_PAGE_SIZE

```
size_t SYSTEM_PAGE_SIZE = 0  [static]
```

Size of the system page.

This static variable holds the size of the system page. It is initialized to 0 and should be updated to the actual size of the system page during program initialization using a system-specific function or method.

Note

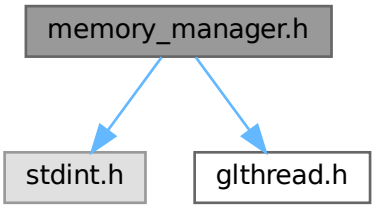
This variable should be initialized to the size of the system page before any memory management operations are performed within the program. The actual size of the system page depends on the underlying operating system and hardware architecture.

5.12 memory_manager.h File Reference

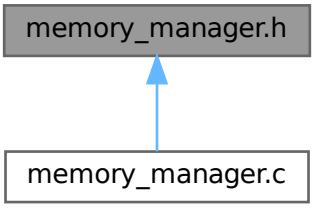
Header file for the Memory Manager module.

```
#include <stdint.h>
#include "glthread.h"
```

Include dependency graph for memory_manager.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct block_meta_data_

  Structure representing metadata for a memory block.
- struct vm_page_

  Structure representing a virtual memory page.
- struct vm_page_family_

  Structure representing a page family in virtual memory.
- struct vm_page_for_families_

  Structure representing a virtual memory page containing families of memory structures.

Macros

- #define MM_MAX_STRUCT_NAME 32
- #define MAX_STRUCT_NAME_LEN 50
- #define MAX_FAMILIES_PER_VM_PAGE

  Maximum number of families that can be stored in a single virtual memory page.
- #define ITERATE_PAGE_FAMILIES_BEGIN(vm_page_for_families_ptr, curr)

  Macro for beginning iteration over page families.
- #define ITERATE_PAGE_FAMILIES_END(vm_page_for_families_ptr, curr)

  Macro marking the end of iteration over families within a virtual memory page.
- #define ITERATE_VM_PAGE_BEGIN(vm_page_family_ptr, curr)

  Macro to iterate over virtual memory pages beginning from the first page of a page family.
- #define ITERATE_VM_PAGE_END(vm_page_family_ptr, curr)

  Macro marking the end of the iteration over virtual memory pages.
- #define ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN(vm_page_ptr, curr)

  Macro to begin iteration over all memory blocks within a virtual memory page.
- #define ITERATE_VM_PAGE_ALL_BLOCKS_END(vm_page_ptr, curr)

  Macro to end iteration over all memory blocks within a virtual memory page.
- #define offset_of(container_structure, field_name) ((size_t)(&((container_structure *)0)->field_name))

  Macro to calculate the offset of a field within a structure.
- #define MM_GET_PAGE_FROM_META_BLOCK(block_meta_data_ptr) ((void *)((char *)(block_meta_data_ptr) - (block_meta_data_ptr)->offset))

  Macro to retrieve the virtual memory page from a block's metadata.
- #define NEXT_META_BLOCK_BY_SIZE(block_meta_data_ptr)

Macro to retrieve the metadata of the next block based on the current block's size.

- #define NEXT_META_BLOCK(block_meta_data_ptr) ((block_meta_data_ptr)->next_block)

  Macro to get the pointer to the next metadata block.
- #define PREV_META_BLOCK(block_meta_data_ptr) ((block_meta_data_ptr)->prev_block)

  Macro to get the pointer to the previous metadata block.
- #define MARK_VM_PAGE_EMPTY(vm_page_t_ptr)

  Macro to mark a virtual memory page as empty.
- #define mm_bind_blocks_for_allocation(allocated_meta_block, free_meta_block)

  Binds metadata blocks for memory allocation.
- #define MAX_PAGE_ALLOCATABLE_MEMORY(units) (mm_max_page_allocatable_memory(units))

  Macro to calculate the maximum allocatable memory for a given number of units.

Typedefs

- typedef struct block_meta_data_ block_meta_data_t

  Structure representing metadata for a memory block.
- typedef struct vm_page_ vm_page_t

  Structure representing a virtual memory page.
- typedef struct vm_page_family_ vm_page_family_t

  Structure representing a page family in virtual memory.
- typedef struct vm_page_for_families_ vm_page_for_families_t

  Structure representing a virtual memory page containing families of memory structures.

Enumerations

- enum vm_bool_t { MM_FALSE = 0 , MM_TRUE = 1 }

  Represents a boolean value.

Functions

- GLTHREAD_TO_STRUCT (glthread_to_block_meta_data, block_meta_data_t, priority_thread_glue, glthread_ptr)

  Macro to declare a conversion function for converting a glthread_t structure to a user-defined structure pointer.
- vm_page_t * allocate_vm_page (vm_page_family_t *vm_page_family)

  Allocates a new virtual memory page for a given page family.
- void mm_vm_page_delete_and_free (vm_page_t *vm_page)

  Deletes and frees a virtual memory page.
- vm_page_family_t * lookup_page_family_by_name (char *struct_name)

  Looks up a page family by its name.
- vm_bool_t mm_is_vm_page_empty (vm_page_t *vm_page)

  Checks if a virtual memory page is empty.
- static void mm_add_free_block_meta_data_to_free_block_list (vm_page_family_t *vm_page_family, block_meta_data_t *free_block)

  Add a free block's metadata to the free block list of a virtual memory page family.
- void mm_print_vm_page_details (vm_page_t *vm_page)

  Prints details of a virtual memory page.
- static void * mm_get_new_vm_page_from_kernel (int units)

  Allocates a new virtual memory page from the kernel.
- static void mm_return_vm_page_to_kernel (void *vm_page, int units)

Returns a virtual memory page to the kernel.

- static void mm_union_free_blocks (block_meta_data_t *first, block_meta_data_t *second)

  Merges two contiguous free memory blocks.
- static uint32_t mm_max_page_allocatable_memory (int units)

  Calculates the maximum allocatable memory within a virtual memory page.
- static int free_blocks_comparison_function (void *_block_meta_data1, void *_block_meta_data2)

  Comparison function for sorting free blocks by block size.
- static block_meta_data_t * mm_get_biggest_free_block_page_family (vm_page_family_t *vm_page_family)

  Retrieves the metadata of the biggest free memory block within a given virtual memory page family.
- static vm_bool_t mm_split_free_data_block_for_allocation (vm_page_family_t *vm_page_family, block_meta_data_t *block_meta_data, uint32_t size)

  Splits a free data block to allocate a portion of it for memory allocation.
- static block_meta_data_t * mm_allocate_free_data_block (vm_page_family_t *vm_page_family, uint32_t req_size)

  Allocates a free data block from the specified page family.
- static vm_page_t * mm_family_new_page_add (vm_page_family_t *vm_page_family)

  Adds a new virtual memory page to the specified page family.
- static int mm_get_hard_internal_memory_frag_size (block_meta_data_t *first, block_meta_data_t *second)

  Calculates the size of hard internal memory fragmentation between two memory blocks.
- static block_meta_data_t * mm_free_blocks (block_meta_data_t *to_be_free_block)

  Frees a memory block and performs merging if necessary.

5.12.1 Detailed Description

Header file for the Memory Manager module.

This file provides declarations for structures, macros, and functions used in the Memory Manager module. The Memory Manager is responsible for managing memory allocation and deallocation, including virtual memory page management, block metadata handling, and allocation algorithms.

5.12.2 Macro Definition Documentation

5.12.2.1 ITERATE_PAGE_FAMILIES_BEGIN

```
#define ITERATE_PAGE_FAMILIES_BEGIN(
                vm_page_for_families_ptr,
                curr )
```

Value:

```
  {                                                                       \
    uint32_t _count = 0;                                                  \
    for (curr =                                                           \
             (vm_page_family_t *)&vm_page_for_families_ptr->vm_page_family[0]; \
         curr->struct_size && _count < MAX_FAMILIES_PER_VM_PAGE;          \
         curr++, _count++) {
```

Macro for beginning iteration over page families.

This macro is used to begin iteration over page families stored within a virtual memory page. It initializes a loop for iterating over page families, using the provided pointers.

Parameters

vm_page_for_families_ptr	Pointer to the virtual memory page for families.
curr	Pointer to the current page family being iterated.

Note

This macro is typically used in conjunction with `ITERATE_PAGE_FAMILIES_END` to iterate over page families stored within a virtual memory page. The loop continues until all page families have been iterated or the maximum number of families per page is reached.

Warning

This macro assumes that `vm_page_for_families_ptr` points to a valid virtual memory page structure containing page families, and `curr` is a valid pointer to iterate over these families. Improper usage may result in undefined behavior.

See also

ITERATE_PAGE_FAMILIES_END

5.12.2.2 ITERATE_PAGE_FAMILIES_END

```
#define ITERATE_PAGE_FAMILIES_END(
              vm_page_for_families_ptr,
              curr )
```

Value:

```
  }                                                                    \
  }
```

Macro marking the end of iteration over families within a virtual memory page.

This macro is used to mark the end of iteration over families within a virtual memory page, which was started with the `ITERATE_PAGE_FAMILIES_BEGIN` macro. It concludes the loop for iterating over page families.

Parameters

vm_page_for_families_ptr	Pointer to the virtual memory page for families.
curr	Pointer to the current family being iterated.

Note

This macro should be used in conjunction with `ITERATE_PAGE_FAMILIES_BEGIN` to properly mark the end of the iteration loop over page families within a virtual memory page.

Warning

The loop for iterating over families within a virtual memory page should be enclosed within curly braces `{}` to ensure proper scoping of loop variables and statements. Improper usage of this macro may lead to compilation errors or unexpected behavior.

See also

ITERATE_PAGE_FAMILIES_BEGIN

5.12.2.3 ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN

```
#define ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN(
                vm_page_ptr,
                curr )
```

Value:

```
  do {                                                                  \
    curr = &(vm_page_ptr->block_meta_data);                             \
    block_meta_data_t *next = NULL;                                     \
    for (; curr != NULL; curr = next) {                                 \
      next = NEXT_META_BLOCK(curr);
```

Macro to begin iteration over all memory blocks within a virtual memory page.

This macro initializes the iteration process over all memory blocks within a given virtual memory page.

Parameters

vm_page_ptr	Pointer to the virtual memory page.
curr	Pointer to hold the current memory block during iteration.

Note

This macro is typically used in memory management systems to iterate over all memory blocks within a virtual memory page. It sets up a loop that traverses through the metadata blocks of each memory block within the page. The iteration begins with the metadata block of the first memory block in the page.

5.12.2.4 ITERATE_VM_PAGE_ALL_BLOCKS_END

```
#define ITERATE_VM_PAGE_ALL_BLOCKS_END(
                vm_page_ptr,
                curr )
```

Value:

```
  }                                                                     \
  }                                                                     \
  while (0)
```

Macro to end iteration over all memory blocks within a virtual memory page.

This macro marks the end of the iteration process over all memory blocks within a virtual memory page.

Parameters

vm_page_ptr	Pointer to the virtual memory page.
curr	Pointer holding the current memory block during iteration.

Note

This macro is used in conjunction with ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN macro to define the end of the iteration loop. It completes the loop setup by ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN, ensuring proper termination of the loop.

5.12.2.5 ITERATE_VM_PAGE_BEGIN

```
#define ITERATE_VM_PAGE_BEGIN(
              vm_page_family_ptr,
              curr )
```

Value:

```
  {                                                                    \
    curr = (vm_page_family_ptr)->first_page;                           \
    vm_page_t *next = NULL;                                            \
    for (; curr != NULL; curr = next) {                                \
      next = curr->next;
```

Macro to iterate over virtual memory pages beginning from the first page of a page family.

This macro allows for iterating over virtual memory pages starting from the first page of a specified page family.

Parameters

vm_page_family_ptr	Pointer to the page family containing the first page.
curr	Pointer variable to hold the current virtual memory page during iteration.

5.12.2.6 ITERATE_VM_PAGE_END

```
#define ITERATE_VM_PAGE_END(
              vm_page_family_ptr,
              curr )
```

Value:

```
  }                                                                    \
  }
```

Macro marking the end of the iteration over virtual memory pages.

This macro marks the end of the iteration over virtual memory pages.

Parameters

vm_page_family_ptr	Pointer to the page family containing the first page.
curr	Pointer variable holding the current virtual memory page.

5.12.2.7 MARK_VM_PAGE_EMPTY

```
#define MARK_VM_PAGE_EMPTY(
              vm_page_t_ptr )
```

Value:

```
do {                                                                    \
    (vm_page_t_ptr)->block_meta_data.next_block = NULL;                 \
    (vm_page_t_ptr)->block_meta_data.prev_block = NULL;                 \
    (vm_page_t_ptr)->block_meta_data.is_free = MM_TRUE;                 \
} while (0)
```

Macro to mark a virtual memory page as empty.

This macro is heavily documented to provide detailed information about its purpose, usage, and behavior.

Parameters

vm_page_t_ptr	Pointer to the virtual memory page to be marked as empty.

This macro is used to reset the state of a virtual memory page, indicating that it contains no allocated memory blocks and is available for reuse. It operates by modifying the metadata associated with the memory blocks within the page.

The macro takes a single parameter:

- `vm_page_t_ptr`: Pointer to the virtual memory page to be marked as empty.

The macro does the following:

- Sets the 'next_block' and 'prev_block' pointers of the block metadata to NULL, indicating that the page does not have any neighboring blocks.
- Sets the 'is_free' flag of the block metadata to MM_TRUE, indicating that the page is free and available for allocation.

Note

This macro should be used judiciously and only when it is certain that the virtual memory page is not in use and can be safely reset. Incorrect usage may lead to memory corruption or undefined behavior.

Warning

It is important to ensure that the 'vm_page_t_ptr' parameter is a valid pointer to a virtual memory page structure. Passing invalid or uninitialized pointers may result in undefined behavior.

Remarks

This macro is typically used in memory management systems as part of memory recycling and allocation routines. It helps maintain memory hygiene by properly managing the state of virtual memory pages.

5.12.2.8 MAX_FAMILIES_PER_VM_PAGE

```
#define MAX_FAMILIES_PER_VM_PAGE
```

Value:

```
  (SYSTEM_PAGE_SIZE - sizeof(struct vm_page_for_families_ *)) /               \
      sizeof(struct vm_page_family_)
```

Maximum number of families that can be stored in a single virtual memory page.

This macro calculates the maximum number of families that can be stored in a single virtual memory page based on the system page size and the sizes of the `vm_page_for_families_t` and `vm_page_family_t` structures. It accounts for the space occupied by the `next` pointer in `vm_page_for_families_t`.

Note

The calculation subtracts the size of the `next` pointer from the total system page size, and then divides the remaining size by the size of a single `vm_page_family_t` structure.

This macro is useful for determining the maximum capacity of a virtual memory page for managing families of memory structures.

5.12.2.9 MAX_PAGE_ALLOCATABLE_MEMORY

```
#define MAX_PAGE_ALLOCATABLE_MEMORY(
              units )   (mm_max_page_allocatable_memory(units))
```

Macro to calculate the maximum allocatable memory for a given number of units.

This macro calculates the maximum allocatable memory for a specified number of units based on the system page size and the offset of the virtual memory page structure.

Parameters

units	Number of units for which memory allocation is requested.

Returns

Maximum allocatable memory in bytes.

Note

This macro is typically used to determine the maximum amount of memory that can be allocated for a given number of units, considering system page constraints and structure offsets within the virtual memory page.

5.12.2.10 MAX_STRUCT_NAME_LEN

```
#define MAX_STRUCT_NAME_LEN 50
```

5.12.2.11 mm_bind_blocks_for_allocation

```
#define mm_bind_blocks_for_allocation(
              allocated_meta_block,
              free_meta_block )
```

Value:

```
  free_meta_block->prev_block = allocated_meta_block;                          \
  free_meta_block->next_block = allocated_meta_block->next_block;              \
  allocated_meta_block->next_block = free_meta_block;                          \
  if (free_meta_block->next_block)                                             \
  free_meta_block->next_block->prev_block = free_meta_block
```

Binds metadata blocks for memory allocation.

This macro is used to bind metadata blocks for memory allocation. It updates the pointers of the allocated and free blocks to maintain the integrity of the memory management system.

Parameters

allocated_meta_block	Pointer to the metadata block of the allocated memory.
free_meta_block	Pointer to the metadata block of the free memory.

Note

This macro is typically used in memory management systems to properly link allocated and free memory blocks. It ensures correct traversal and management of memory blocks, maintaining the coherence of the memory allocation process.

5.12.2.12 MM_GET_PAGE_FROM_META_BLOCK

```
#define MM_GET_PAGE_FROM_META_BLOCK(
              block_meta_data_ptr )  ((void *)((char *)(block_meta_data_ptr) - (block_meta_↩
data_ptr)->offset))
```

Macro to retrieve the virtual memory page from a block's metadata.

This macro retrieves the virtual memory page associated with a given block's metadata.

Parameters

block_meta_data_ptr	Pointer to the block's metadata.

Returns

Pointer to the virtual memory page.

5.12.2.13 MM_MAX_STRUCT_NAME

```
#define MM_MAX_STRUCT_NAME 32
```

< System includes < External includes

5.12.2.14 NEXT_META_BLOCK

```
#define NEXT_META_BLOCK(
              block_meta_data_ptr ) ((block_meta_data_ptr)->next_block)
```

Macro to get the pointer to the next metadata block.

This macro is used to obtain the pointer to the next metadata block given a pointer to the current metadata block.

Parameters

block_meta_data_ptr	Pointer to the current metadata block.

Returns

Pointer to the next metadata block.

Note

This macro is typically used in memory management systems where metadata blocks are used to manage memory allocation. It allows for efficient traversal of the metadata blocks linked list, enabling operations such as coalescing adjacent free memory blocks or iterating over allocated memory blocks.

5.12.2.15 NEXT_META_BLOCK_BY_SIZE

```
#define NEXT_META_BLOCK_BY_SIZE(
              block_meta_data_ptr )
```

Value:

```
  ((block_meta_data_t *)((char *)(block_meta_data_ptr + 1) +                  \
                         (block_meta_data_ptr)->block_size))
```

Macro to retrieve the metadata of the next block based on the current block's size.

This macro calculates the pointer to the metadata of the next block by adding the size of the current block to the pointer to the current block's metadata.

Parameters

block_meta_data_ptr	Pointer to the current block's metadata.

Returns

Pointer to the metadata of the next block.

Note

This macro is commonly used in memory management systems where metadata blocks are used to manage memory allocation. It allows for efficient traversal of the memory blocks, enabling operations such as coalescing adjacent free memory blocks or iterating over allocated memory blocks.

Warning

The behavior of this macro depends on the assumption that the next block starts immediately after the current block in memory. Ensure that the memory layout and block sizes are correctly managed to avoid undefined behavior.

5.12.2.16 offset_of

```
#define offset_of(
              container_structure,
              field_name )  ((size_t)(&((container_structure *)0)->field_name))
```

Macro to calculate the offset of a field within a structure.

This macro calculates the byte offset of a specified field within a structure. It is often used in low-level programming to access structure members at specific memory locations.

Parameters

container_structure	The name of the structure containing the field.
field_name	The name of the field whose offset is being calculated.

Returns

The byte offset of the field within the structure.

Note

This macro uses the pointer arithmetic to calculate the offset.

5.12.2.17 PREV_META_BLOCK

```
#define PREV_META_BLOCK(
              block_meta_data_ptr ) ((block_meta_data_ptr)->prev_block)
```

Macro to get the pointer to the previous metadata block.

This macro is used to obtain the pointer to the previous metadata block given a pointer to the current metadata block.

Parameters

block_meta_data_ptr	Pointer to the current metadata block.

Returns

Pointer to the previous metadata block.

Note

This macro is typically used in memory management systems where metadata blocks are used to manage memory allocation. It allows for efficient traversal of the metadata blocks linked list, allowing operations such as merging adjacent free memory blocks or finding neighboring blocks.

5.12.3 Typedef Documentation

5.12.3.1 block_meta_data_t

```
typedef struct block_meta_data_ block_meta_data_t
```

Structure representing metadata for a memory block.

The `block_meta_data_t` structure represents metadata for a memory block. It includes information such as whether the block is free or allocated, its size, pointers to the previous and next blocks (if applicable), and the offset within the memory region.

5.12.3.2 vm_page_family_t

```
typedef struct vm_page_family_ vm_page_family_t
```

Structure representing a page family in virtual memory.

This structure maintains information about a page family in virtual memory, including the name of the structure, its size, a pointer to the most recent virtual memory page in use, and a priority list of free memory blocks.

5.12.3.3 vm_page_for_families_t

```
typedef struct vm_page_for_families_ vm_page_for_families_t
```

Structure representing a virtual memory page containing families of memory structures.

5.12.3.4 vm_page_t

```
typedef struct vm_page_ vm_page_t
```

Structure representing a virtual memory page.

This structure represents a virtual memory page used in memory management systems. It contains metadata for managing memory blocks within the page, as well as the actual memory region allocated for storing data blocks.

5.12.4 Enumeration Type Documentation

5.12.4.1 vm_bool_t

```
enum vm_bool_t
```

Represents a boolean value.

The `vm_bool_t` type represents a boolean value, which can have one of two states: `VM_TRUE` or `VM_FALSE`. It is used to store boolean values in the program.

Enumerator

MM_FALSE	Represents the false state.
MM_TRUE	Represents the true state.

5.12.5 Function Documentation

5.12.5.1 allocate_vm_page()

```
vm_page_t * allocate_vm_page (
            vm_page_family_t * vm_page_family )
```

Allocates a new virtual memory page for a given page family.

This function allocates a new virtual memory page for the specified page family. It initializes the metadata and pointers associated with the page and inserts the page into the linked list of pages belonging to the page family.

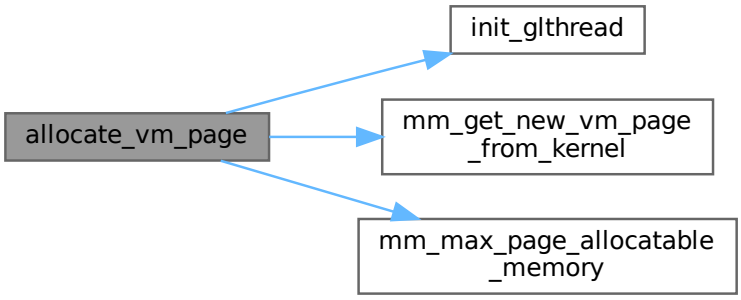
Parameters

vm_page_family	Pointer to the page family for which the page is being allocated.

Returns

Pointer to the newly allocated virtual memory page.

Here is the call graph for this function:



Here is the caller graph for this function:

5.12.5.2 free_blocks_comparison_function()

```
static int free_blocks_comparison_function (
            void * _block_meta_data1,
            void * _block_meta_data2 )  [static]
```

Comparison function for sorting free blocks by block size.

This function compares two block_meta_data_t objects based on their block sizes. It is intended to be used as a comparison function for sorting free blocks in descending order of block size.

Parameters

_block_meta_data1	Pointer to the first block_meta_data_t object.
_block_meta_data2	Pointer to the second block_meta_data_t object.

Returns

An integer value representing the result of the comparison:

- If the block size of _block_meta_data1 is greater than that of _block_meta_data2, the function returns -1.
- If the block size of _block_meta_data1 is less than that of _block_meta_data2, the function returns 1.
- If the block sizes are equal, the function returns 0.

5.12.5.3 GLTHREAD_TO_STRUCT()

```
GLTHREAD_TO_STRUCT (
            glthread_to_block_meta_data ,
            block_meta_data_t ,
            priority_thread_glue ,
            glthread_ptr  )
```

Macro to declare a conversion function for converting a glthread_t structure to a user-defined structure pointer.

This macro simplifies the process of declaring a conversion function that takes a glthread_t pointer and returns a pointer to a user-defined structure. It is particularly useful when you have a glthread_t structure embedded within a user-defined structure and need to access the user-defined data.

Parameters

fn_name	The name of the conversion function to be declared.
struct_type	The type of the user-defined structure.
glthread_member	The name of the glthread_t member within the user-defined structure.
glthread_ptr	The name of the glthread_t pointer variable.

Example usage: Suppose we have a user-defined structure named block_meta_data_t that contains a glthread↩ _t member named priority_thread_glue. To declare a conversion function named glthread_to_block_meta_data to convert a glthread_t pointer to a block_meta_data_t pointer, we use the following declaration:

GLTHREAD_TO_STRUCT(glthread_to_block_meta_data, block_meta_data_t, priority_thread_glue, glthread_ptr);

Now, we can use glthread_to_block_meta_data to convert glthread_t pointers to block_meta_data_t pointers and access the metadata associated with memory blocks.

5.12.5.4 lookup_page_family_by_name()

```
vm_page_family_t * lookup_page_family_by_name (
            char * struct_name )
```

Looks up a page family by its name.

This function iterates over all virtual memory pages hosting page families and returns a pointer to the page family object identified by the given struct_name. If no such page family object is found, it returns NULL.

Parameters

struct_name	The name of the page family to look up.

Returns

Pointer to the page family object if found, otherwise NULL.

Note

This function should be used to retrieve a page family object by its name after the page families have been registered and initialized using the appropriate functions and macros provided by the memory manager.

See also

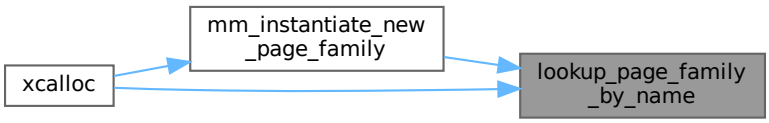
mm_init

MM_REG_STRUCT

vm_page_for_families_t

vm_page_family_t

Here is the caller graph for this function:



5.12.5.5 mm_add_free_block_meta_data_to_free_block_list()

```
static void mm_add_free_block_meta_data_to_free_block_list (
            vm_page_family_t * vm_page_family,
            block_meta_data_t * free_block )  [static]
```

Add a free block's metadata to the free block list of a virtual memory page family.

This function adds the metadata of a free block to the free block list of a virtual memory page family. The block metadata is inserted into the free block list in descending order of block size, based on the comparison function `free_blocks_comparison_function`.

Parameters

vm_page_family	Pointer to the virtual memory page family to which the free block metadata will be added.
free_block	Pointer to the block_meta_data_t structure representing the metadata of the free block to be added to the free block list.

Note

This function assumes that the `is_free` flag of the `free_block` structure is set to MM_TRUE. An assertion will trigger if this condition is not met.

5.12.5.6 mm_allocate_free_data_block()

```
static block_meta_data_t * mm_allocate_free_data_block (
            vm_page_family_t * vm_page_family,
            uint32_t req_size )  [static]
```

Allocates a free data block from the specified page family.

This function attempts to allocate a free data block of the requested size from the specified page family. It first checks if there is a sufficiently large free block available within the page family. If not, it adds a new page to the page family to satisfy the allocation request. If successful, it splits the free block to allocate the requested memory and returns a pointer to the allocated block's metadata.

Parameters

vm_page_family	Pointer to the page family from which to allocate the data block.
req_size	The size of the data block to allocate.

Returns

A pointer to the allocated block's metadata if successful, or NULL if the allocation fails.

Note

This function assumes that the specified page family has been properly initialized and that the requested size is within the maximum allocatable memory per page. It utilizes the mm_family_new_page_add and mm_split↩ _free_data_block_for_allocation functions to add new pages and split free blocks for allocation, respectively.

5.12.5.7 mm_family_new_page_add()

```
static vm_page_t * mm_family_new_page_add (
            vm_page_family_t * vm_page_family )  [static]
```

Adds a new virtual memory page to the specified page family.

This function adds a new virtual memory page to the specified page family. It first allocates a new page using the allocate_vm_page function and then adds the page to the page family. Additionally, it treats the new page as one free block and adds its metadata to the free block list of the page family.

Parameters

vm_page_family	Pointer to the page family to which the new page will be added.

Returns

A pointer to the newly added virtual memory page if successful, or NULL if allocation fails.

Note

This function assumes that the page family has been properly initialized and that the allocate_vm_page function is available for allocating new pages. It also relies on the mm_add_free_block_meta_data_to_free_↩ block_list function to add the metadata of the new page to the free block list of the page family.

5.12.5.8 mm_free_blocks()

```
static block_meta_data_t * mm_free_blocks (
            block_meta_data_t * to_be_free_block )  [static]
```

Frees a memory block and performs merging if necessary.

This function frees a memory block represented by the given `to_be_free_block` parameter. It also handles merging of adjacent free blocks if present.

Parameters

to_be_free_block	The block to be freed.

Returns

A pointer to the freed block or NULL if the hosting page becomes empty.

Note

The function assumes that `to_be_free_block` is not NULL and its is_free flag is set to MM_FALSE (indicating it's not already free). MM_H_

5.12.5.9 mm_get_biggest_free_block_page_family()

```
static block_meta_data_t * mm_get_biggest_free_block_page_family (
            vm_page_family_t * vm_page_family )  [inline], [static]
```

Retrieves the metadata of the biggest free memory block within a given virtual memory page family.

This function retrieves the metadata of the biggest free memory block within a specified virtual memory page family. It utilizes a priority list to maintain the biggest free block at the head of the list.

Parameters

vm_page_family	Pointer to the virtual memory page family for which the biggest free block is to be retrieved.

Returns

Pointer to the metadata of the biggest free memory block within the page family. If no such block exists (i.e., the priority list is empty), it returns NULL.

Note

This function is typically used in memory management systems to efficiently locate the largest available free block within a page family, which can then be used for memory allocation.

5.12.5.10 mm_get_hard_internal_memory_frag_size()

```
static int mm_get_hard_internal_memory_frag_size (
            block_meta_data_t * first,
            block_meta_data_t * second )  [static]
```

Calculates the size of hard internal memory fragmentation between two memory blocks.

This function calculates the size of hard internal memory fragmentation between two memory blocks. Hard internal memory fragmentation occurs when there is unused space between the end of the first memory block and the start of the second memory block.

Parameters

first	Pointer to the first memory block.
second	Pointer to the second memory block.

Returns

The size of hard internal memory fragmentation between the two memory blocks.

5.12.5.11 mm_get_new_vm_page_from_kernel()

```
static void * mm_get_new_vm_page_from_kernel (
            int units )  [static]
```

Allocates a new virtual memory page from the kernel.

This function allocates a new virtual memory page from the kernel and returns a pointer to the allocated memory block. It uses the `mmap()` system call to request the allocation of memory from the kernel.

Parameters

units	The number of memory pages to allocate.

Returns

A pointer to the allocated memory block, or NULL if the allocation fails.

Note

The size of the allocated memory block is determined by multiplying the specified number of units by the system page size (defined by `SYSTEM_PAGE_SIZE`).

Warning

This function should be used with caution as it interacts directly with the kernel to allocate memory. Improper use or misuse of this function can lead to memory leaks or system instability.

See also

man mmap()

5.12.5.12 mm_is_vm_page_empty()

```
vm_bool_t mm_is_vm_page_empty (
            vm_page_t * vm_page )
```

Checks if a virtual memory page is empty.

This function determines whether a virtual memory page is empty based on its metadata.

Parameters

vm_page	Pointer to the virtual memory page to be checked.

Returns

- MM_TRUE if the page is empty.
- MM_FALSE if the page is not empty or if the input pointer is NULL.

Note

A virtual memory page is considered empty if all the following conditions are met:

- The 'next_block' pointer in the block metadata is NULL, indicating no next block.
- The 'prev_block' pointer in the block metadata is NULL, indicating no previous block.
- The 'is_free' flag in the block metadata is set to MM_TRUE, indicating the page is free.

Warning

It is important to ensure that the 'vm_page' parameter is a valid pointer to a virtual memory page structure. Passing invalid or uninitialized pointers may result in undefined behavior.

Here is the caller graph for this function:

5.12.5.13 mm_max_page_allocatable_memory()

```
static uint32_t mm_max_page_allocatable_memory (
            int units )  [inline], [static]
```

Calculates the maximum allocatable memory within a virtual memory page.

This function computes the maximum amount of memory that can be allocated within a virtual memory page, given the number of units specified.

Parameters

units	The number of memory units to be allocated.

Returns

The maximum allocatable memory size in bytes.

Note

This function takes into account the size of the virtual memory page and subtracts the offset of the page memory within the vm_page_t structure to determine the available memory for allocation. It is typically used in memory management systems to ensure proper allocation of memory within virtual memory pages.

5.12.5.14 mm_print_vm_page_details()

```
void mm_print_vm_page_details (
            vm_page_t * vm_page )
```

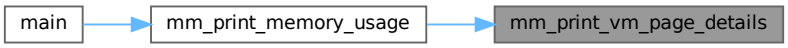
Prints details of a virtual memory page.

This function prints detailed information about a virtual memory page, including its next and previous pointers, page family name, and information about each block within the page.

Parameters

vm_page	Pointer to the virtual memory page.

Here is the caller graph for this function:



5.12.5.15 mm_return_vm_page_to_kernel()

```
static void mm_return_vm_page_to_kernel (
            void * vm_page,
            int units )  [static]
```

Returns a virtual memory page to the kernel.

This function returns a virtual memory page previously allocated from the kernel back to the kernel. It uses the `munmap()` system call to release the memory.

Parameters

vm_page	A pointer to the memory block to be returned to the kernel.
units	The number of memory pages to return to the kernel.

Note

The size of the memory block to be returned is determined by multiplying the specified number of units by the system page size (defined by `SYSTEM_PAGE_SIZE`).

Warning

This function should be used with caution as it interacts directly with the kernel to release memory. Improper use or misuse of this function can lead to memory leaks or system instability.

See also

man munmap()

5.12.5.16 mm_split_free_data_block_for_allocation()

```
static vm_bool_t mm_split_free_data_block_for_allocation (
            vm_page_family_t * vm_page_family,
            block_meta_data_t * block_meta_data,
            uint32_t size )  [static]
```

Splits a free data block to allocate a portion of it for memory allocation.

This function splits a free data block to allocate a portion of it for memory allocation. It checks various cases to determine how the block should be split and whether additional metadata blocks need to be created. After splitting, it updates the metadata of the original block and, if necessary, creates new metadata blocks for the remaining free space.

Parameters

vm_page_family	Pointer to the page family associated with the data block.
block_meta_data	Pointer to the metadata of the free data block to be split.
size	Size of the portion of the block to be allocated.

Returns

MM_TRUE if the block is successfully split and allocated, MM_FALSE otherwise.

Note

This function assumes that the provided block is free and that the size argument specifies a valid size for memory allocation. It relies on the mm_bind_blocks_for_allocation function to establish the link between metadata blocks after splitting.

5.12.5.17 mm_union_free_blocks()

```
static void mm_union_free_blocks (
            block_meta_data_t * first,
            block_meta_data_t * second )  [static]
```

Merges two contiguous free memory blocks.

This function merges two contiguous free memory blocks into a single block. The function assumes that both blocks are free and contiguous.

Parameters

first	Pointer to the first free memory block.
second	Pointer to the second free memory block.

Note

This function is typically used in memory management systems to optimize memory usage by consolidating adjacent free memory blocks.

5.12.5.18 mm_vm_page_delete_and_free()

```
void mm_vm_page_delete_and_free (
            vm_page_t * vm_page )
```

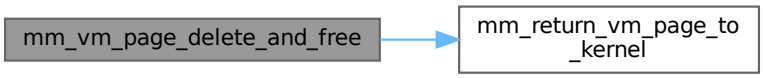
Deletes and frees a virtual memory page.

This function deletes and frees a virtual memory page. It removes the page from the linked list of pages belonging to its page family and deallocates the memory associated with the page.

Parameters

vm_page	Pointer to the virtual memory page to be deleted and freed.

Here is the call graph for this function:



Here is the caller graph for this function:

5.13 memory_manager.h

Go to the documentation of this file.

```
00001 /*************************************************************************/
00002 /******* Author     : Mahmoud Abdelraouf Mahmoud *****************/
00003 /******* Date       : 8 Apr 2023                 *****************/
00004 /******* Version    : 0.1                        *****************/
00005 /******* File Name  : MemeoryManager.h           *****************/
00006 /*************************************************************************/
00007
00018 #ifndef MM_H_
00019 #define MM_H_
00020
00021 //-----------------< Includes section -----------------/
00023 #include <stdint.h>
00025 #include "glthread.h"
00026
00027 //-----------------< Macros section -----------------/
00028 #define MM_MAX_STRUCT_NAME 32
00029 #define MAX_STRUCT_NAME_LEN 50
00030
00031 //-----------------< user defined data type section -----------------/
00039 typedef enum {
00040   MM_FALSE = 0,
00041   MM_TRUE = 1
00042 } vm_bool_t;
00043
00052 typedef struct block_meta_data_ {
00053   vm_bool_t is_free;
00054   uint32_t block_size;
00055   uint32_t offset;
00056   struct block_meta_data_
00057       *prev_block;
00058   struct block_meta_data_ *next_block;
00059   glthread_t priority_thread_glue;
00061 } block_meta_data_t;
00062
00091 GLTHREAD_TO_STRUCT(glthread_to_block_meta_data, block_meta_data_t,
00092                    priority_thread_glue, glthread_ptr);
00093
```

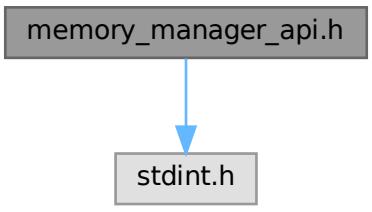
```
00101 typedef struct vm_page_ {
00102   struct vm_page_ *next;
00103   struct vm_page_ *prev;
00104   struct vm_page_family_
00105       *pg_family;
00106   block_meta_data_t block_meta_data;
00108   char page_memory[0];
00109 } vm_page_t;
00110
00118 typedef struct vm_page_family_ {
00119   char struct_name[MM_MAX_STRUCT_NAME];
00120   uint32_t struct_size;
00121   vm_page_t *first_page;
00122   glthread_t free_block_priority_list_head;
00124 } vm_page_family_t;
00125
00130 typedef struct vm_page_for_families_ {
00131   struct vm_page_for_families_
00132       *next;
00133   vm_page_family_t vm_page_family[0];
00135 } vm_page_for_families_t;
00136
00149 vm_page_t *allocate_vm_page(vm_page_family_t *vm_page_family);
00150
00160 void mm_vm_page_delete_and_free(vm_page_t *vm_page);
00161
00162 //-----------------< Public functions interface -----------------/
00183 vm_page_family_t *lookup_page_family_by_name(char *struct_name);
00184
00212 vm_bool_t mm_is_vm_page_empty(vm_page_t *vm_page);
00213
00232 static void
00233 mm_add_free_block_meta_data_to_free_block_list(vm_page_family_t *vm_page_family,
00234                                                block_meta_data_t *free_block);
00235
00245 void mm_print_vm_page_details(vm_page_t *vm_page);
00246
00247 //-----------------< Function-like macro section -----------------/
00264 #define MAX_FAMILIES_PER_VM_PAGE                                        \
00265   (SYSTEM_PAGE_SIZE - sizeof(struct vm_page_for_families_ *)) /         \
00266       sizeof(struct vm_page_family_)
00267
00291 #define ITERATE_PAGE_FAMILIES_BEGIN(vm_page_for_families_ptr, curr)     \
00292   {                                                                     \
00293     uint32_t _count = 0;                                                \
00294     for (curr =                                                         \
00295              (vm_page_family_t *)&vm_page_for_families_ptr->vm_page_family[0]; \
00296          curr->struct_size && _count < MAX_FAMILIES_PER_VM_PAGE;        \
00297          curr++, _count++) {
00298
00322 #define ITERATE_PAGE_FAMILIES_END(vm_page_for_families_ptr, curr)       \
00323   }                                                                     \
00324   }
00325
00338 #define ITERATE_VM_PAGE_BEGIN(vm_page_family_ptr, curr)                 \
00339   {                                                                     \
00340     curr = (vm_page_family_ptr)->first_page;                            \
00341     vm_page_t *next = NULL;                                             \
00342     for (; curr != NULL; curr = next) {                                 \
00343       next = curr->next;
00344
00354 #define ITERATE_VM_PAGE_END(vm_page_family_ptr, curr)                   \
00355   }                                                                     \
00356   }
00357
00374 #define ITERATE_VM_PAGE_ALL_BLOCKS_BEGIN(vm_page_ptr, curr)             \
00375   do {                                                                  \
00376     curr = &(vm_page_ptr->block_meta_data);                             \
00377     block_meta_data_t *next = NULL;                                     \
00378     for (; curr != NULL; curr = next) {                                 \
00379       next = NEXT_META_BLOCK(curr);
00380
00395 #define ITERATE_VM_PAGE_ALL_BLOCKS_END(vm_page_ptr, curr)               \
00396   }                                                                     \
00397   }                                                                     \
00398   while (0)
00399
00414 #define offset_of(container_structure, field_name)                      \
00415   ((size_t)(&((container_structure *)0)->field_name))
00416
00426 #define MM_GET_PAGE_FROM_META_BLOCK(block_meta_data_ptr)                \
00427   ((void *)((char *)(block_meta_data_ptr) - (block_meta_data_ptr)->offset))
00428
00449 #define NEXT_META_BLOCK_BY_SIZE(block_meta_data_ptr)                    \
00450   ((block_meta_data_t *)((char *)(block_meta_data_ptr + 1) +            \
00451                          (block_meta_data_ptr)->block_size))
00452
```

```
00469 #define NEXT_META_BLOCK(block_meta_data_ptr) ((block_meta_data_ptr)->next_block)
00470
00486 #define PREV_META_BLOCK(block_meta_data_ptr) ((block_meta_data_ptr)->prev_block)
00487
00527 #define MARK_VM_PAGE_EMPTY(vm_page_t_ptr)                                      \
00528   do {                                                                         \
00529     (vm_page_t_ptr)->block_meta_data.next_block = NULL;                        \
00530     (vm_page_t_ptr)->block_meta_data.prev_block = NULL;                        \
00531     (vm_page_t_ptr)->block_meta_data.is_free = MM_TRUE;                        \
00532   } while (0)
00533
00550 #define mm_bind_blocks_for_allocation(allocated_meta_block, free_meta_block)   \
00551   free_meta_block->prev_block = allocated_meta_block;                          \
00552   free_meta_block->next_block = allocated_meta_block->next_block;              \
00553   allocated_meta_block->next_block = free_meta_block;                          \
00554   if (free_meta_block->next_block)                                             \
00555   free_meta_block->next_block->prev_block = free_meta_block
00556
00572 #define MAX_PAGE_ALLOCATABLE_MEMORY(units)                                     \
00573   (mm_max_page_allocatable_memory(units))
00574
00575 //-----------------< Private functions interfacce -----------------/
00597 static void *mm_get_new_vm_page_from_kernel(int units);
00598
00619 static void mm_return_vm_page_to_kernel(void *vm_page, int units);
00620
00633 static void mm_union_free_blocks(block_meta_data_t *first,
00634                                  block_meta_data_t *second);
00635
00652 static inline uint32_t mm_max_page_allocatable_memory(int units);
00653
00670 static int free_blocks_comparison_function(void *_block_meta_data1,
00671                                            void *_block_meta_data2);
00672
00692 static inline block_meta_data_t *
00693 mm_get_biggest_free_block_page_family(vm_page_family_t *vm_page_family);
00694
00719 static vm_bool_t
00720 mm_split_free_data_block_for_allocation(vm_page_family_t *vm_page_family,
00721                                         block_meta_data_t *block_meta_data,
00722                                         uint32_t size);
00723
00747 static block_meta_data_t *
00748 mm_allocate_free_data_block(vm_page_family_t *vm_page_family,
00749                             uint32_t req_size);
00771 static vm_page_t *mm_family_new_page_add(vm_page_family_t *vm_page_family);
00772
00787 static int mm_get_hard_internal_memory_frag_size(block_meta_data_t *first,
00788                                                  block_meta_data_t *second);
00789
00804 static block_meta_data_t *mm_free_blocks(block_meta_data_t *to_be_free_block);
00805
00806 #endif
```

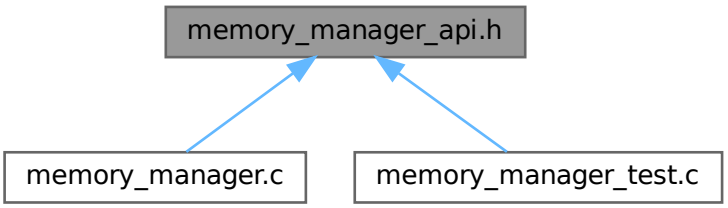
5.14 memory_manager_api.h File Reference

Header file for the Memory Manager API.

`#include <stdint.h>`
Include dependency graph for memory_manager_api.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define MM_REG_STRUCT(struct_name) (mm_instantiate_new_page_family(#struct_name, sizeof(struct↩ _name)))

  Registers a memory structure for page family instantiation.
- #define XCALLOC(units, struct_name) (xcalloc(#struct_name, units))

  Macro for allocating memory for multiple instances of a structure and initializing them to zero.
- #define XFREE(ptr) (xfree(ptr))

  Macro for freeing memory using a custom deallocation function.

Functions

- void mm_init ()

  Initializes the memory manager.
- void mm_instantiate_new_page_family (char ∗struct_name, uint32_t struct_size)

  Instantiates a new page family for a memory structure.
- void ∗ xcalloc (char ∗struct_name, int units)

  Allocates and initializes memory for an array of structures.
- void xfree (void ∗app_data)

  Frees memory allocated by the memory manager.
- void mm_print_registered_page_families ()

  Prints all registered page families.
- void mm_print_memory_usage (char ∗struct_name)

  Prints memory usage details related to the memory manager.
- void mm_print_block_usage ()

  Prints information about the memory block usage.

5.14.1 Detailed Description

Header file for the Memory Manager API.

This file contains declarations for the Memory Manager API functions. The Memory Manager API provides functions for initializing the memory manager, managing page families, allocating and freeing memory, and printing memory usage details.

5.14.2 Macro Definition Documentation

5.14.2.1 MM_REG_STRUCT

```
#define MM_REG_STRUCT(
              struct_name )  (mm_instantiate_new_page_family(#struct_name, sizeof(struct_↩
name)))
```

Registers a memory structure for page family instantiation.

This macro facilitates the registration of a memory structure for page family instantiation within the memory manager. It takes the name of the structure (`struct_name`) as a parameter and uses the `#` operator to stringify it, which is then passed to the `mm_instantiate_new_page_family()` function along with the size of the structure.

Parameters

struct_name	The name of the memory structure to be registered.

Note

This macro should be used to register each memory structure before it is instantiated as a page family within the memory manager. It ensures proper initialization of the memory management system and enables efficient allocation and management of memory pages.

See also

mm_instantiate_new_page_family()

5.14.2.2 XCALLOC

```
#define XCALLOC(
              units,
              struct_name ) (xcalloc(#struct_name, units))
```

Macro for allocating memory for multiple instances of a structure and initializing them to zero.

This macro simplifies the process of allocating memory for multiple instances of a structure and initializing them to zero. It takes the number of units and the name of the structure as input parameters.

Parameters

units	The number of instances of the structure to allocate memory for.
struct_name	The name of the structure for which memory is to be allocated.

Returns

A pointer to the allocated memory, initialized to zero, or NULL if allocation fails.

5.14.2.3 XFREE

```
#define XFREE(
              ptr ) (xfree(ptr))
```

Macro for freeing memory using a custom deallocation function.

This macro is used for freeing memory using a custom deallocation function specified by the user. The macro takes a pointer to the memory to be freed as its argument and passes it to the custom deallocation function xfree().

Parameters

ptr	Pointer to the memory to be freed. UAPI_MM_H_

5.14.3 Function Documentation

5.14.3.1 mm_init()

```
void mm_init ( )
```

Initializes the memory manager.

This function initializes the memory manager. It sets up necessary configurations and parameters for memory management operations within the program. It specifically determines the system page size using the `getpagesize()` system call and assigns it to the global variable `SYSTEM_PAGE_SIZE`.

Note

This function should be called before any memory management operations are performed within the program. It is typically called at the beginning of the program execution to ensure proper initialization of memory management functionalities.

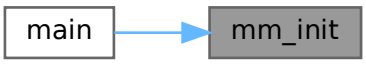
Warning

This function relies on the `getpagesize()` system call to determine the system page size. Therefore, it may not be portable across all platforms. It is primarily intended for use in Unix-like systems where `getpagesize()` is available.

See also

man getpagesize()

Here is the caller graph for this function:



5.14.3.2 mm_instantiate_new_page_family()

```
void mm_instantiate_new_page_family (
            char * struct_name,
            uint32_t struct_size )
```

Instantiates a new page family for a memory structure.

This function creates a new page family for a memory structure identified by its name and size. It allocates memory for the page family and adds it to the existing virtual memory pages if necessary. Each page family can contain multiple memory structures of the same type.

Parameters

struct_name	The name of the memory structure.
struct_size	The size of the memory structure.

Note

If the size of the memory structure exceeds the system page size, an error message is printed, and the function returns without creating the page family.

This function maintains a linked list of virtual memory pages (`first_vm_page_for_families`) to store the page families. If there are no existing pages, it allocates a new page and initializes it with the first page family. If the existing pages are full, it allocates a new page and adds it to the beginning of the linked list.

If a page family with the same name already exists, an assertion error is triggered, indicating a conflict in page family instantiation.

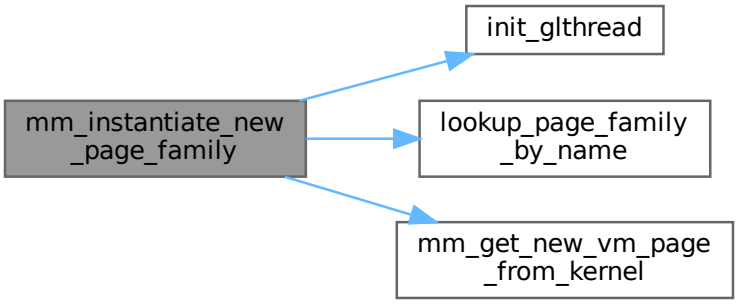
Warning

This function relies on the `mm_get_new_vm_page_from_kernel()` function to allocate memory from the kernel for the page family. Improper use or misuse of this function can lead to memory leaks or system instability.

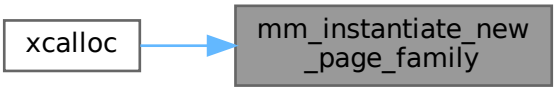
See also

mm_get_new_vm_page_from_kernel()

Here is the call graph for this function:



Here is the caller graph for this function:



5.14.3.3 mm_print_block_usage()

```
void mm_print_block_usage ( )
```

Prints information about the memory block usage.

This function iterates through all virtual memory pages and their associated memory block families to print information about the memory block usage, including the total block count, free block count, occupied block count, and application memory usage. Here is the caller graph for this function:

5.14.3.4 mm_print_memory_usage()

```
void mm_print_memory_usage (
            char * struct_name )
```

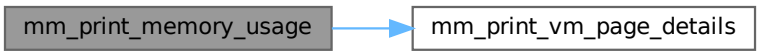
Prints memory usage details related to the memory manager.

This function prints information about the memory usage of the memory manager, including details of each virtual memory page family and the total memory being used. Optionally, it can filter the output by a specific structure name.

Parameters

struct_name	Optional parameter to filter the output by a specific structure name.

Here is the call graph for this function:



Here is the caller graph for this function:

5.14.3.5 mm_print_registered_page_families()

```
void mm_print_registered_page_families ( )
```

Prints all registered page families.

This function prints all page families that have been registered with the Linux Memory Manager. It iterates over all virtual memory pages hosting page families and prints information about each page family, including its name and size.

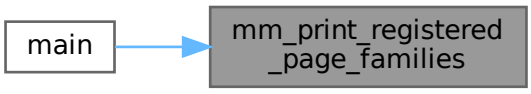
Note

This function should be invoked after the application has performed registration for all its structures using the `MM_REG_STRUCT` macro. It relies on the `first_vm_page_for_families` global variable, which maintains a linked list of virtual memory pages containing page families.

See also

MM_REG_STRUCT

Here is the caller graph for this function:



5.14.3.6 xcalloc()

```
void * xcalloc (
            char * struct_name,
            int units )
```

Allocates and initializes memory for an array of structures.

This function allocates memory for an array of structures of the specified type and initializes the memory to zero. It first looks up the page family associated with the specified structure name to determine the size of the structure. Then, it checks if the requested memory size exceeds the maximum allocatable memory per page. If the allocation is successful, it initializes the allocated memory to zero and returns a pointer to the allocated memory.

Parameters

struct_name	The name of the structure type for which memory is to be allocated.
units	The number of structures to allocate.

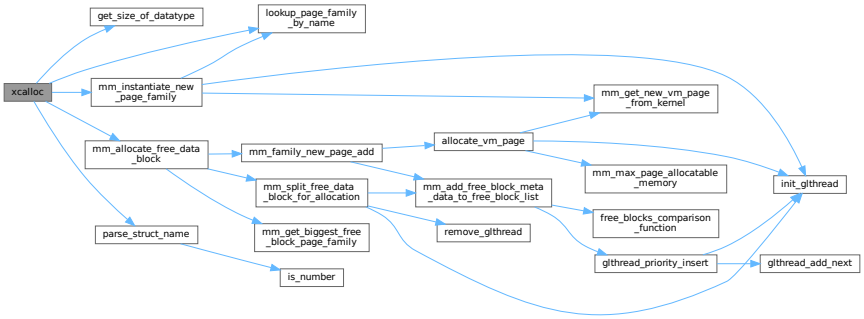
Returns

A pointer to the allocated memory if successful, or NULL if the allocation fails.

Note

This function assumes that the specified structure type has been registered with the Memory Manager using the mm_register_structure function. It also assumes that the specified structure type has a corresponding page family registered with the Memory Manager.

Here is the call graph for this function:



5.14.3.7 xfree()

```
void xfree (
            void * app_data )
```

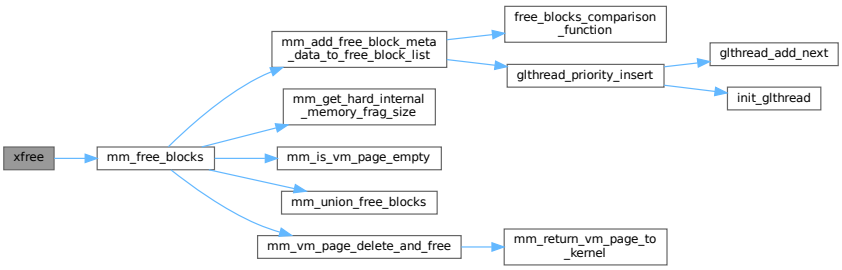
Frees memory allocated by the memory manager.

This function is used to free memory that was previously allocated by the memory manager. It takes a pointer to the memory to be freed as its argument. The pointer is adjusted to point to the block metadata, and then it is passed to the memory manager's free blocks function.

Parameters

app_data	Pointer to the memory to be freed.

Here is the call graph for this function:



5.15 memory_manager_api.h

Go to the documentation of this file.

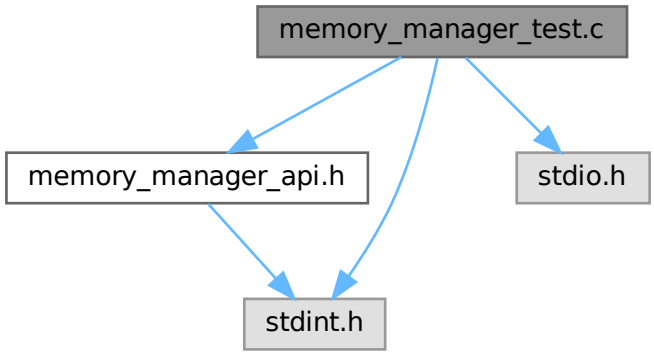
```
00001 /*********************************************************************/
00002 /******* Author    : Mahmoud Abdelraouf Mahmoud *********************/
00003 /******* Date      : 8 Apr 2023                 *********************/
00004 /******* Version   : 0.1                        *********************/
00005 /******* File Name : memory_manager_api.h       *********************/
00006 /*********************************************************************/
00007
00018 #ifndef UAPI_MM_H_
00019 #define UAPI_MM_H_
00020
00021 //-----------------< Includes section -----------------/
00022 #include <stdint.h>
00023
00024 //-----------------< Public functions interface section -----------------/
00046 void mm_init();
00047
00078 void mm_instantiate_new_page_family(char *struct_name, uint32_t struct_size);
00079
00102 void *xcalloc(char *struct_name, int units);
00103
00114 void xfree(void *app_data);
00115
00131 void mm_print_registered_page_families();
00144 void mm_print_memory_usage(char *struct_name);
00145
00154 void mm_print_block_usage();
00155
00156 //-----------------< Function-like macro section -----------------/
00175 #define MM_REG_STRUCT(struct_name)                                     \
00176   (mm_instantiate_new_page_family(#struct_name, sizeof(struct_name)))
00177
00193 #define XCALLOC(units, struct_name) (xcalloc(#struct_name, units))
00194
00204 #define XFREE(ptr) (xfree(ptr))
00205
00206 #endif
```

5.16 memory_manager_test.c File Reference

Test file for Memory Manager functionality.

```
#include "memory_manager_api.h"
#include <stdint.h>
#include <stdio.h>
```

Include dependency graph for memory_manager_test.c:



Data Structures

- struct emp_

  Structure representing an employee.
- struct student_

  Structure representing a student.

Typedefs

- typedef struct emp_ emp_t

  Structure representing an employee.
- typedef struct student_ student_t

  Structure representing a student.

Functions

- int main (int argc, char **argv)

  The main function.

5.16.1 Detailed Description

Test file for Memory Manager functionality.

This file is used to test the Memory Manager module on a Linux system. It includes the main function to initialize necessary components, register structure types, and print registered page families. Additionally, it contains sample allocations to test memory management operations.

5.16.2 Typedef Documentation

5.16.2.1 emp_t

```
typedef struct emp_ emp_t
```

Structure representing an employee.

This structure defines the attributes of an employee, including their name and employee ID.

5.16.2.2 student_t

```
typedef struct student_ student_t
```

Structure representing a student.

This structure defines the attributes of a student, including their name, roll number, and subject marks. Additionally, it contains a pointer to the next student in a linked list.

5.16.3 Function Documentation

5.16.3.1 main()

```
int main (
            int argc,
            char ** argv )
```

The main function.

This function serves as the entry point to the program. It initializes necessary components, registers structure types, and prints registered page families.

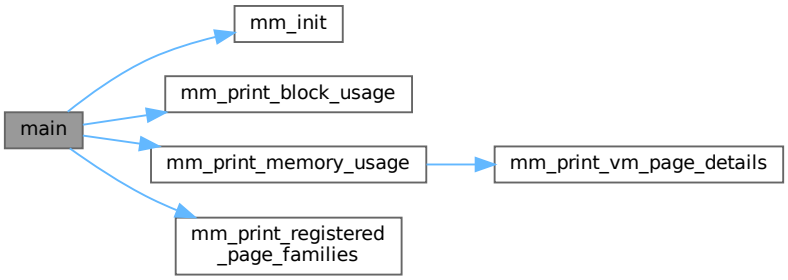
Parameters

argc	The number of command-line arguments.
argv	An array of command-line arguments.

Returns

An integer indicating the exit status of the program.

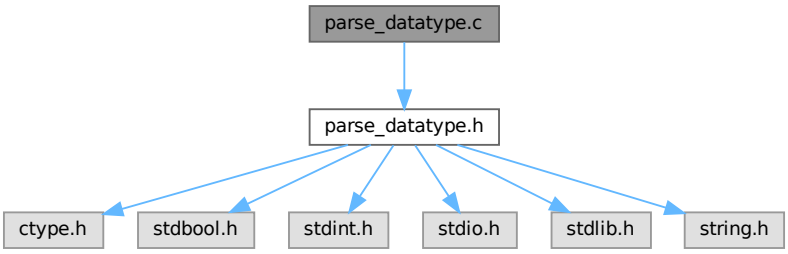
Here is the call graph for this function:



5.17 parse_datatype.c File Reference

Extracts a data type name from a string containing the sizeof operator.

`#include "parse_datatype.h"`
Include dependency graph for parse_datatype.c:



Functions

- char ∗ parse_struct_name (char ∗struct_name, char ∗buffer, uint8_t ∗error_flag)

  Parses the name of a struct from a string.
- static bool is_number (const char ∗str)

5.17.1 Detailed Description

Extracts a data type name from a string containing the sizeof operator.

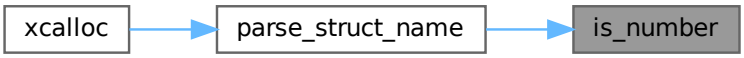
This program demonstrates how to extract a data type name from a string containing the sizeof operator using the sscanf function in C.

5.17.2 Function Documentation

5.17.2.1 is_number()

```
static bool is_number (
            const char * str )  [static]
```

Here is the caller graph for this function:



5.17.2.2 parse_struct_name()

```
char * parse_struct_name (
            char * struct_name,
            char * buffer,
            uint8_t * error_flag )
```

Parses the name of a struct from a string.

This function extracts the name of a struct from a string formatted as "sizeof(datatype)" and stores it in the provided buffer. If the string does not match this format, it checks if the string represents a number and sets the error flag accordingly.

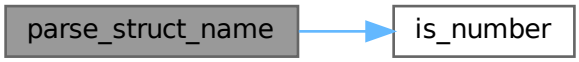
Parameters

struct_name	The string containing the sizeof expression.
buffer	Pointer to a buffer where the extracted struct name or integer value will be stored. If the struct name is extracted successfully, it will be stored in this buffer. If the struct name is a number, the integer value will be stored in this buffer.
error_flag	Pointer to a uint8_t variable to indicate the status of the struct name. It can have the following values: • 0: No error, struct name extracted successfully. • 1: Error in the format of the struct name, not in the form "sizeof(datatype)". • 2: The struct name is actually a number, and the integer value is stored in the buffer.

Returns

If the struct name is extracted successfully or it's a number, returns "int". If an error occurs, returns NULL.

Here is the call graph for this function:



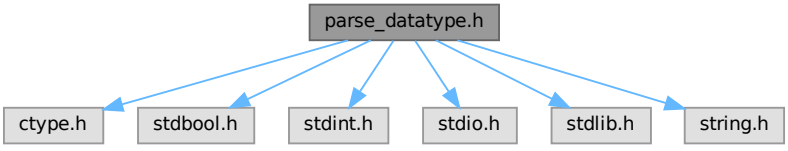
Here is the caller graph for this function:

5.18 parse_datatype.h File Reference

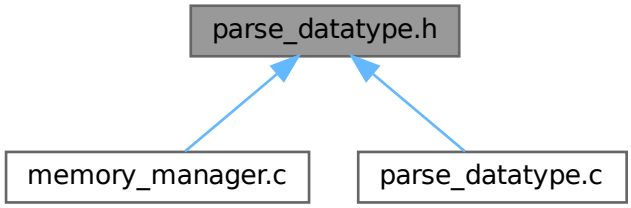
Header file for parsing data type names.

```
#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Include dependency graph for parse_datatype.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define MAX_STRUCT_NAME_LEN 50
  Maximum length of the data type name.

Functions

- char * parse_struct_name (char *struct_name, char *buffer, uint8_t *error_flag)
  Parses the name of a struct from a string.
- static bool is_number (const char *str)
  Checks if a string represents a number.

5.18.1 Detailed Description

Header file for parsing data type names.

This file contains declarations for functions related to parsing data type names from strings.

5.18.2 Macro Definition Documentation

5.18.2.1 MAX_STRUCT_NAME_LEN

```
#define MAX_STRUCT_NAME_LEN 50
```

Maximum length of the data type name.

Defines the maximum length allowed for a data type name.

5.18.3 Function Documentation

5.18.3.1 is_number()

```
static bool is_number (
            const char * str )  [static]
```

Checks if a string represents a number.

This function checks if the provided string represents a number. It allows for digits (0-9) and an optional decimal point. It ignores leading and trailing whitespace.

Parameters

str	The string to check.

Returns

true if the string represents a number, false otherwise. PARSE_DATATYPE_H_

5.18.3.2 parse_struct_name()

```
char * parse_struct_name (
            char * struct_name,
            char * buffer,
            uint8_t * error_flag )
```

Parses the name of a struct from a string.

This function extracts the name of a struct from a string formatted as "sizeof(datatype)" and stores it in the provided buffer. If the string does not match this format, it checks if the string represents a number and sets the error flag accordingly.

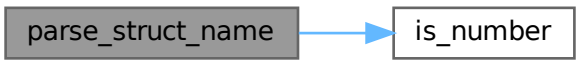
Parameters

struct_name	The string containing the sizeof expression.
buffer	Pointer to a buffer where the extracted struct name or integer value will be stored. If the struct name is extracted successfully, it will be stored in this buffer. If the struct name is a number, the integer value will be stored in this buffer.
error_flag	Pointer to a uint8_t variable to indicate the status of the struct name. It can have the following values: • 0: No error, struct name extracted successfully. • 1: Error in the format of the struct name, not in the form "sizeof(datatype)". • 2: The struct name is actually a number, and the integer value is stored in the buffer.

Returns

If the struct name is extracted successfully or it's a number, returns "int". If an error occurs, returns NULL.

Here is the call graph for this function:



Here is the caller graph for this function:

5.19 parse_datatype.h

Go to the documentation of this file.

```
00001 /*****************************************************************/
00002 /******* Author    : Mahmoud Abdelraouf Mahmoud *****************/
00003 /******* Date      : 8 Apr 2023                 *****************/
00004 /******* Version   : 0.1                        *****************/
00005 /******* File Name : parse_datatype.h           *****************/
00006 /*****************************************************************/
00007
00016 #ifndef PARSE_DATATYPE_H_
00017 #define PARSE_DATATYPE_H_
00018
00019 #include <ctype.h>
00020 #include <stdbool.h>
00021 #include <stdint.h>
00022 #include <stdio.h>
00023 #include <stdlib.h>
00024 #include <string.h>
00025
00032 #define MAX_STRUCT_NAME_LEN 50
00033
00057 char *parse_struct_name(char *struct_name, char *buffer, uint8_t *error_flag);
00058
00069 static bool is_number(const char *str);
00070
00071 #endif
```

Index