Project 7

Contiguous Memory Allocation

Introduction

In this project, we will implement a contiguous memory allocation algorithm. The allocator will be able to:

- Allocate contiguous memory from a large memory block
- Free allocated memory
- Automatically merge adjacent free blocks
- Report memory usage statistics
- Switch between first-fit, best-fit, and worst-fit allocation strategies
- Align memory allocation to a specified byte boundary

We also provide a command-line interface to interact with the allocator and test its functionality.

```
allocator > RQ mem1 13418 F
allocator > RQ mem2 89424 W
allocator > RQ mem3 92325 B
allocator > RQ mem4 19341 F
allocator > RQ mem5 12441 F
allocator > STAT
allocator memory pointer: 0x7fffffefdb70
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  16383]       mem1     Size = 16384
  Address [  16384: 106495]       mem2     Size = 90112
  Address [ 106496: 200703]       mem3     Size = 94208
  Address [ 200704: 221183]       mem4     Size = 20480
  Address [ 221184: 237567]       mem5     Size = 16384
  Address [ 237568:1048575]     Unused     Size = 811008
allocator > RL mem2
allocator > RL mem4
allocator > STAT
allocator memory pointer: 0x7fffffefdb70
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  16383]       mem1     Size = 16384
  Address [  16384: 106495]     Unused     Size = 90112
  Address [ 106496: 200703]       mem3     Size = 94208
  Address [ 200704: 221183]     Unused     Size = 20480
  Address [ 221184: 237567]       mem5     Size = 16384
  Address [ 237568:1048575]     Unused     Size = 811008
allocator > RQ mem6 20480 W
allocator > RQ mem7 20480 F
allocator > RQ mem8 20480 B
allocator > STAT
allocator memory pointer: 0x7fffffefdb70
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  16383]       mem1     Size = 16384
  Address [  16384:  36863]       mem7     Size = 20480
  Address [  36864: 106495]     Unused     Size = 69632
  Address [ 106496: 200703]       mem3     Size = 94208
  Address [ 200704: 221183]       mem8     Size = 20480
  Address [ 221184: 237567]       mem5     Size = 16384
  Address [ 237568: 258047]       mem6     Size = 20480
  Address [ 258048:1048575]     Unused     Size = 790528
allocator >
```

Figure 1: Test using command-line interface

Overview

The memory allocator is implemented in C using object-oriented programming techniques. The allocator is consisted of the following components:

- Alloclist: A singly linked list to store allocated memory blocks, the nodes of the list are kept in ascending order of the starting address of the memory block.
- Freelist: A cross-linked list to store free memory blocks, the nodes of the list are kept both in ascending order of the starting address of the memory block and in ascending order of the size of the memory block.
- Policy: A set of functions to implement first-fit, best-fit, and worst-fit allocation strategies.
- Allocator: A structure to integrate the above components and provide a unified interface to the user.

The structure of the allocator is as follows:

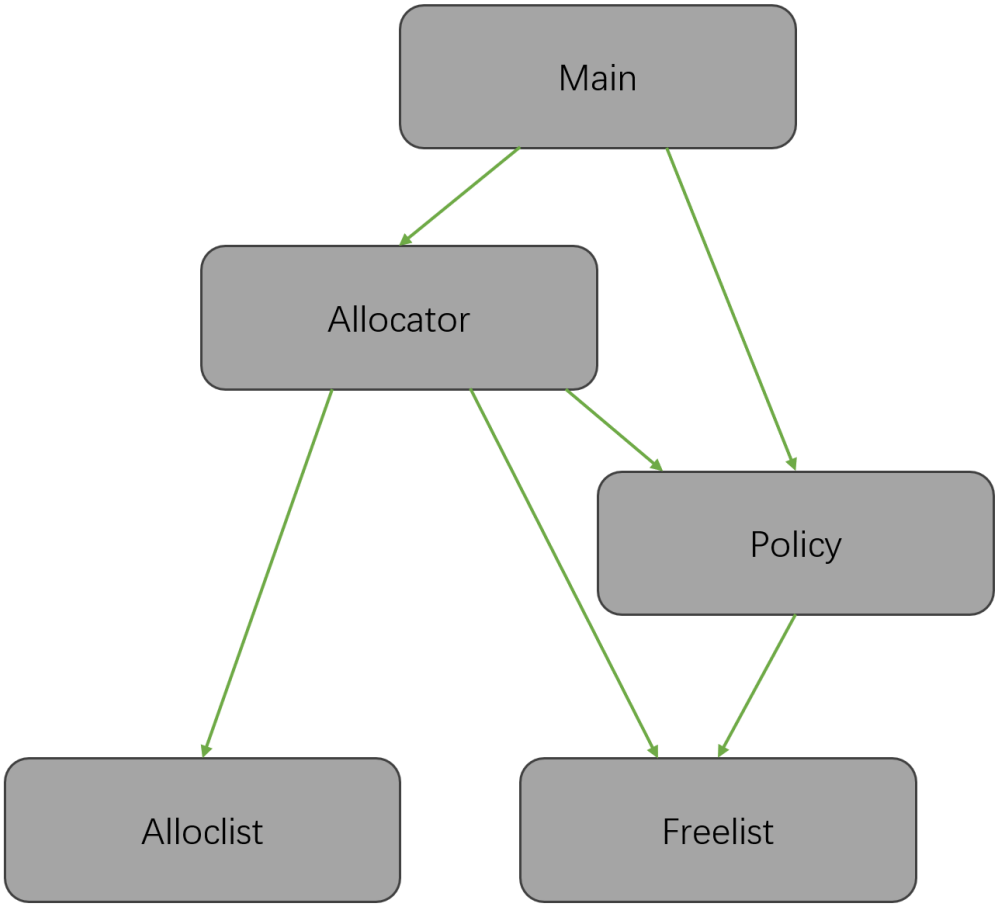

Figure 2: Memory allocator structure

Implementation

Alloclist

The alloclist is a singly linked list to store allocated memory blocks. The nodes store information about the starting address, size and the name of the allocated memory block. The nodes are kept in ascending order of the starting address of the memory block in order to report memory usage statistics in human-friendly format.

The data structure of the alloclist is:

```
/// @brief Information about an allocated memory block
typedef struct alloc_info
{
    char     *name;
    ptrdiff_t address;
    size_t    size;
} alloc_info_t;

/// @brief Node in the alloclist
typedef struct alloclist_node
{
    alloc_info_t           info;
    struct alloclist_node *next;
} alloclist_node_t;

/// @brief A linked list of allocated memory blocks
typedef struct alloclist
{
    alloclist_node_t *head;
} alloclist_t;
```

The main operations of the alloclist are:

```
/// @brief Allocate a new memory block
void alloclist_alloc(alloclist_t *list, const char *name, ptrdiff_t
address, size_t size);

/// @brief Free a memory block
alloc_info_t alloclist_free(alloclist_t *list, const char *name);

/// @brief Get the address of the memory block
/// @return Address of the memory block, or -1 if not found
ptrdiff_t alloclist_address(alloclist_t *list, const char *name);
```

The allocation function creates a new node in the alloclist and inserts it in the correct position based on the starting address of the memory block.

The free function finds the node with the specified name in the alloclist by traversing the alloclist and removes it. The function returns the information of the removed node.

The address function returns the starting address of the memory block with the specified name by traversing the alloclist.

Freelist

The freelist is a cross-linked list to store free memory blocks. The nodes store information about the starting address and size of the free memory block. The nodes are kept in ascending order of the starting address of the memory block in one direction (prev, next) and in ascending order of the size of the memory block in the other direction (smaller, larger). This organization allows the allocator to locate free memory blocks efficiently and merge adjacent free blocks automatically.

The data structure of the freelist is:

```
/// @brief A node in a freelist
typedef struct freelist_node
{
    ptrdiff_t             address;
    size_t                size;
    struct freelist_node *prev;
    struct freelist_node *next;
    struct freelist_node *smaller;
    struct freelist_node *larger;
} freelist_node_t;

/// @brief An ordered cross linked list for managing free memory
typedef struct freelist
{
    freelist_node_t *node;
    ptrdiff_t        address;
    size_t           size;
} freelist_t;
```

The main operations of the freelist are:

```
/// @brief Allocate memory from the node
void freelist_alloc(freelist_t *list, freelist_node_t *node, size_t
size);
```

```
/// @brief Free memory to the freelist
void freelist_free(freelist_t *list, ptrdiff_t address, size_t
size);

/// @brief Internal function: try to merge the node with the
address adjacent nodes
static void freelist_internal_merge(freelist_t *list,
freelist_node_t *node);
```

The allocation function first checks if the size of the free memory block is larger than the requested size. If so, the function splits the free memory block into two parts: one for the allocated memory block and the other for the remaining free memory block. The function moves the node along the size direction to keep the order of memory size in the freelist. Otherwise the free block is allocated entirely and the node is removed from the freelist.

The free function creates a new node with the specified address and size and inserts it into the freelist by address order. The function then tries to merge the new node with the adjacent nodes to keep the freelist compact. The size order is maintained by the internal merge function.

The internal merge function tries to merge the node with the adjacent nodes in the address direction. The function recursively merges the node with the smaller node and the larger node if they are adjacent. The function also moves the node along the size direction to keep the order of memory size in the freelist.

Policy

The policy is a set of functions to implement first-fit, best-fit, and worst-fit allocation strategies. The functions take the freelist and the requested size as input and return the node that satisfies the allocation strategy. The functions are implemented as follows:

```
ptrdiff_t first_fit(freelist_t *list, size_t size)
{
    /* Search for the first node that can fit the allocation */
    freelist_node_t *node = list->node->next;
    while (node != list->node)
    {
        if (node->size >= size)
        {
            ptrdiff_t address = node->address;
```

```
            freelist_alloc(list, node, size);
            return address;
        }
        node = node->next;
    }
    return -1;
}

ptrdiff_t best_fit(freelist_t *list, size_t size)
{
    /* Search for the smallest node that can fit the allocation */
    freelist_node_t *node = list->node->larger;
    while (node != list->node)
    {
        if (node->size >= size)
        {
            ptrdiff_t address = node->address;
            freelist_alloc(list, node, size);
            return address;
        }
        node = node->larger;
    }
    return -1;
}

ptrdiff_t worst_fit(freelist_t *list, size_t size)
{
    /* Search for the largest node that can fit the allocation */
    freelist_node_t *node = list->node->smaller;
    if (node->size >= size)
    {
        ptrdiff_t address = node->address;
        freelist_alloc(list, node, size);
        return address;
    }
    return -1;
}
```

The first-fit function and the best-fit function traverse the freelist in a certain direction. This allows the functions to find the suitable node quickly because they do not need to traverse the entire freelist. The worst-fit function, on the other hand, only needs to check the largest node in the freelist because it always allocates the largest free memory block.

Allocator

The allocator is a structure to integrate the above components and provide a unified interface to the user. It also pads the allocated memory to align to a specified byte boundary. The allocator structure is:

```
/// @brief A memory allocator that manages a block of allocated
memory
typedef struct allocator
{
    void       *memory;
    size_t      size;
    size_t      page_size;
    alloclist_t alloclist;
    freelist_t  freelist;
} allocator_t;
```

The main operations of the allocator are:

```
/// @brief Allocate a new memory block
/// @param alloc_policy Policy function to use for allocation
/// @return Address of the memory block, or NULL if allocation
///         failed
void *allocator_alloc(allocator_t *allocator, const char *name,
size_t size, policy_t alloc_policy);

/// @brief Free a memory block, if multiple blocks have the same
///        name, free the first one
/// @param allocator The allocator to free from
/// @param name Name of the memory block
void allocator_free(allocator_t *allocator, const char *name);

/// @brief Print memory allocation information
/// @param allocator The allocator to print
void allocator_print(allocator_t *allocator);

/// @brief Round up a size to the multiple of allocator page size
static size_t allocator_internal_roundup(allocator_t *allocator,
size_t size);
```

The allocation function first rounds up the requested size to the multiple of the allocator page size. The function then calls the policy function to find the suitable node in the freelist. If the policy function returns a valid address, the function updates the alloclist and returns the address. Otherwise, the function returns NULL.

The free function calls the free function of the alloclist to remove the node with the specified name. If the node is found, the function calls the free function of the freelist to insert the free memory block into the freelist.

The print function prints the memory allocation information in human-friendly format. The function traverses the alloclist and the freelist to report the usage statistics by ascending order of the starting address.

The internal roundup function rounds up the size to the multiple of the allocator page size. The function is implemented in bit manipulation to improve performance, but it may not work correctly if the page size is not a power of 2. The implementation is as follows:

```
static size_t allocator_internal_roundup(allocator_t *allocator,
size_t size)
{
    /* This works iff allocator->page_size is a power of 2 */
    return (size + allocator->page_size - 1) &
           ~(allocator->page_size - 1);
}
```

Command-line Interface

The command-line interface provides a way to interact with the allocator and test its functionality. It parses the command-line arguments and calls the corresponding functions of the allocator. It also provides a help message to guide the user on how to use the allocator. The interface responds to the following commands:

- `RQ <block_name> <block_size> <allocate_policy: F/B/W>`: Allocate a memory block with the specified name, size, and allocation policy.
- `RL <block_name>`: Free the memory block with the specified name.
- `STAT`: Print memory allocation information.
- `HELP`: Print help message.
- `EXIT`: Exit the program.

Correctness

We have tested the allocator using various test cases to ensure its correctness. The following test cases are used to verify the functionality of the allocator:

- The allocator can round up the size to the multiple of the page size correctly.

```
allocator > RQ roundup1 1 F
allocator > RQ roundup2 4096 F
allocator > RQ roundup3 73825 F
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:   4095]   roundup1     Size = 4096
  Address [   4096:   8191]   roundup2     Size = 4096
  Address [   8192:  86015]   roundup3     Size = 77824
  Address [  86016:1048575]     Unused     Size = 962560
allocator >
```

Figure 3: Test rounding up the size

- The allocator can allocate and free memory blocks correctly.

```
allocator > RQ mem1 82375 F
allocator > RQ mem2 2785 W
allocator > RQ mem3 92752 B
allocator > RQ mem4 97325 F
allocator > RQ mem5 235 F
allocator > RL mem2
allocator > RL mem4
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  86015]       mem1     Size = 86016
  Address [  86016:  90111]     Unused     Size = 4096
  Address [  90112: 184319]       mem3     Size = 94208
  Address [ 184320: 282623]     Unused     Size = 98304
  Address [ 282624: 286719]       mem5     Size = 4096
  Address [ 286720:1048575]     Unused     Size = 761856
allocator > RQ too_large 2735829 F
Memory allocation failed
allocator > RL not_exist
Memory block "not_exist" does not exist
allocator > RQ mem1 245 F
Memory block "mem1" already exists
allocator >
```

Figure 4: Test allocating and freeing memory blocks

- The allocator can merge adjacent free blocks automatically.

```
allocator > RQ mem1 285 F
allocator > RQ mem2 237529 F
allocator > RQ mem3 7998 F
allocator > RQ mem4 8425 F
allocator > RQ mem5 3089 F
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:   4095]       mem1     Size = 4096
  Address [   4096: 241663]       mem2     Size = 237568
  Address [ 241664: 249855]       mem3     Size = 8192
  Address [ 249856: 262143]       mem4     Size = 12288
  Address [ 262144: 266239]       mem5     Size = 4096
  Address [ 266240:1048575]     Unused     Size = 782336
allocator > RL mem2
allocator > RL mem3
allocator > RL mem4
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:   4095]       mem1     Size = 4096
  Address [   4096: 262143]     Unused     Size = 258048
  Address [ 262144: 266239]       mem5     Size = 4096
  Address [ 266240:1048575]     Unused     Size = 782336
allocator > RL mem1
allocator > RL mem5
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:1048575]     Unused     Size = 1048576
allocator >
```

Figure 5: Test merging adjacent free blocks

- The allocator can switch between first-fit, best-fit, and worst-fit allocation strategies correctly. Suppose we have the following memory allocation:

```
allocator > RQ mem1 23587 F
allocator > RQ mem2 29485 F
allocator > RQ mem3 92735 F
allocator > RQ mem4 20480 F
allocator > RQ mem5 9358 F
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  24575]       mem1     Size = 24576
  Address [  24576:  57343]       mem2     Size = 32768
  Address [  57344: 151551]       mem3     Size = 94208
  Address [ 151552: 172031]       mem4     Size = 20480
  Address [ 172032: 184319]       mem5     Size = 12288
  Address [ 184320:1048575]     Unused     Size = 864256
allocator > RL mem2
allocator > RL mem4
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  24575]       mem1     Size = 24576
  Address [  24576:  57343]     Unused     Size = 32768
  Address [  57344: 151551]       mem3     Size = 94208
  Address [ 151552: 172031]     Unused     Size = 20480
  Address [ 172032: 184319]       mem5     Size = 12288
  Address [ 184320:1048575]     Unused     Size = 864256
allocator >
```

Figure 6: Current memory allocation

Then we allocate a new memory block sized 20480 bytes. If we use the first-fit policy, it should allocate in the first `Unused` block. If we use the best-fit policy, it should allocate in the second `Unused` block. If we use the worst-fit policy, it should allocate in the third `Unused` block. The result is as follows:

```
allocator > RQ first 20480 F
allocator > RQ best 20480 B
allocator > RQ worst 20480 W
allocator > STAT
allocator memory pointer: 0x7fffffefdb40
allocator size: 1048576 bytes
allocator page size: 4096 bytes
allocator allocation list:
  Address [      0:  24575]       mem1     Size = 24576
  Address [  24576:  45055]      first     Size = 20480
  Address [  45056:  57343]     Unused     Size = 12288
  Address [  57344: 151551]       mem3     Size = 94208
  Address [ 151552: 172031]       best     Size = 20480
  Address [ 172032: 184319]       mem5     Size = 12288
  Address [ 184320: 204799]      worst     Size = 20480
  Address [ 204800:1048575]     Unused     Size = 843776
allocator >
```

Figure 7: Memory allocation after allocating a new block

It can be seen that the allocator allocates the memory block correctly based on the allocation policy.

Conclusion

In this project, we have implemented a contiguous memory allocation algorithm. The allocator can allocate contiguous memory from a large memory block, free allocated memory, automatically merge adjacent free blocks, report memory usage statistics, switch between first-fit, best-fit, and worst-fit allocation strategies, and align memory allocation to a specified byte boundary.

We have also provided a command-line interface to interact with the allocator and test its functionality. The allocator has been tested using various test cases to ensure its correctness. The allocator is a useful tool for managing memory allocation efficiently and effectively.

The allocator is an important component in computer systems and can be used in various applications such as operating systems, databases, and virtual machines. The efficiency and correctness of the allocator are crucial for the performance and reliability of the system. By implementing and testing the allocator, we have gained valuable experience in memory management and object-oriented programming techniques.

In this project, we have learned how to implement a contiguous memory allocation algorithm and how to use object-oriented programming techniques to organize the code. We have also learned how to test the allocator using various test cases to ensure its correctness. The allocator is a valuable tool for managing memory allocation in a large memory block.

The project may optimized for performance by using more efficient data structures and algorithms like hash tables or binary search. The project may also be extended to support more advanced features like memory protection, compaction, and garbage collectio in the future.