Custom Heap Allocator

Author : Mehrdad Varmaziyar,Kiarash Kamranikia
BASU_UNIVERSITY

https://github.com/Mehrdad386/Heap_Allocater.git

Project Report

Custom Heap Manager with Mark-and-Sweep Garbage Collection in C

---

1. Objective

The objective of this project is to design and implement a **custom heap memory manager** in the C programming language, operating on raw memory provided by the operating system. The system replaces standard dynamic memory functions (malloc, free) with user-defined equivalents and demonstrates how memory allocation mechanisms work internally.

In addition to basic memory allocation and deallocation, the project aims to:

- Manage memory using explicit metadata structures.
- Reduce external fragmentation through chunk splitting and merging.
- Detect common heap-related errors such as double free and heap corruption.
- Implement a **Mark-and-Sweep Garbage Collection (GC)** algorithm.
- Introduce a basic **Heap Spraying detection** mechanism by limiting active allocations.

---

2. Introduction

Dynamic memory management is a fundamental component of low-level software systems such as operating systems, language runtimes, and embedded systems. In the C programming language, memory management is explicit and manual, giving the programmer full control but also exposing programs to serious reliability and security risks.

Heap-related vulnerabilities—including heap corruption, memory leaks, and use-after-free errors—are among the most common causes of program crashes and security exploits. Understanding how heap allocators function internally is therefore essential for systems programmers and security engineers.

This project provides a hands-on implementation of a simplified heap allocator, including a basic garbage collector. The design closely follows classical allocator designs while remaining simple enough to be easily understood and extended.

---

3. Overall Architecture and Design

3.1 Heap Representation

The heap is represented as a **singly linked list of memory chunks**. Each chunk contains metadata followed by a payload region returned to the user.

```
+---------------------+----------------------+
| chunk_t (metadata)  | user payload         |
+---------------------+----------------------+
```

3.2 Allocation Policy

The allocator uses the **First-Fit** strategy:

- The heap is traversed from the beginning.
- The first free chunk large enough to satisfy the request is selected.
- If possible, the chunk is split to avoid wasting memory.

3.3 Deallocation Policy

When memory is freed:

- The chunk is marked as unused.
- Adjacent free chunks are merged (coalescing) to reduce fragmentation.

---

4. Core Data Structures

- ### 4.1 Chunk Metadata Structure

```
struct chunk_t {
    uint32_t size;
    uint8_t inuse;
    uint8_t marked;
    uint32_t magic;
    struct chunk_t *next;
};
```

Field Descriptions

- **size**
  Size of the payload in bytes.
- **inuse**
  Indicates whether the chunk is allocated (1) or free (0).
- **marked**
  Used by the garbage collector during the mark phase.
- **magic**
  Stores a constant value (0xDEADBEEF) to detect heap corruption or invalid pointers.
- **next**
  Pointer to the next chunk in the heap.

This structure is critical because all memory operations depend on correctly interpreting and maintaining chunk metadata.

4.2 Heap Metadata Structure

```
struct heap_t {
    struct chunk_t *start;
    uint32_t avail;
    uint32_t active_allocs;
};
```

 start: Pointer to the first chunk in the heap.

 avail: Tracks available payload memory.

 active_allocs: Counts active allocations and supports heap spraying detection.

---

5. Heap Initialization (hinit)

Purpose

The hinit function prepares a raw memory region to be used as a managed heap.

Function Prototype

```c
int hinit(struct heap_t *h, void *mem, uint32_t size);
```

Key Code

```c
if (h == NULL || mem == NULL) {
    errno = EFAULT;
    return -1;
}

if (h->start != NULL) {
    errno = EBUSY;
    return -1;
}

if (size <= sizeof(struct chunk_t)) {
    errno = ENOMEM;
    return -1;
}
```

Chunk Initialization

```c
first = (struct chunk_t *)mem;
first->size = size - sizeof(struct chunk_t);
first->inuse = 0;
first->marked = 0;
first->magic = CHUNK_MAGIC;
first->next = NULL;

h->start = first;
h->avail = first->size;
h->active_allocs = 0;
```

Explanation

- Ensures valid parameters.
- Prevents heap reinitialization.
- Creates a single large free chunk.
- Initializes heap metadata.

---

6. Memory Allocation (halloc)

Purpose

Allocates memory from the heap and returns a pointer to the payload.

Function Prototype

```c
void *halloc(struct heap_t *h, size_t size);
```

Heap Spraying Detection

```c
if (h->active_allocs >= MAX_ACTIVE_ALLOCS) {
    errno = EOVERFLOW;
    return NULL;
}
```

Parameter Validation

```c
if (h == NULL || size == 0) {
    errno = EINVAL;
    return NULL;
}
```

First-Fit Traversal and Splitting

```c
if (!current->inuse && current->size >= size) {
    if (current->size >= size + sizeof(struct chunk_t) + 1) {

        new_chunk = (struct chunk_t *)(
            (uint8_t *)current + sizeof(struct chunk_t) + size
        );

        new_chunk->size = current->size - size - sizeof(struct chunk_t);
        new_chunk->inuse = 0;
        new_chunk->marked = 0;
        new_chunk->magic = CHUNK_MAGIC;
        new_chunk->next = current->next;

        current->size = size;
        current->next = new_chunk;
    }

    current->inuse = 1;
    h->avail -= current->size;
    current->magic = CHUNK_MAGIC;
    h->active_allocs++;

    return (uint8_t *)current + sizeof(struct chunk_t);
}
```

Explanation

- Traverses heap using First-Fit.
- Splits large chunks to minimize fragmentation.
- Updates heap statistics.
- Returns a pointer to usable memory.

---

7. Memory Deallocation (hfree)

Purpose

Releases previously allocated memory back to the heap.

Function Prototype

```c
void hfree(struct heap_t *h, void *ptr);
```

Pointer Validation and Metadata Retrieval

```c
if (h == NULL || ptr == NULL) {
    errno = EINVAL;
    return;
}

chunk = (struct chunk_t *)((uint8_t *)ptr - sizeof(struct chunk_t));
```

Double Free and Corruption Detection

```c
if (!chunk->inuse) {
    errno = EINVAL;
    return;
}

if (chunk->magic != CHUNK_MAGIC) {
    errno = EFAULT;
    return;
}
```

Freeing and Coalescing

```c
chunk->inuse = 0;
h->avail += chunk->size;

current = h->start;
while (current != NULL) {
    if (!current->inuse &&
        current->next &&
        !current->next->inuse) {

        current->size += sizeof(struct chunk_t) + current->next->size;
        current->next = current->next->next;
        continue;
    }
    current = current->next;
}
```

Explanation

- Prevents invalid frees.
- Detects heap corruption.
- Merges adjacent free chunks to reduce fragmentation.

---

8. Garbage Collection (Mark-and-Sweep)

8.1 Root Registration

```c
void gc_register_root(void *ptr) {
    if (gc_root_count < MAX_ROOTS)
        gc_roots[gc_root_count++] = ptr;
}
```

```c
void gc_unregister_root(void *ptr) {
    for (uint32_t i = 0; i < gc_root_count; i++) {
        if (gc_roots[i] == ptr) {
            gc_roots[i] = gc_roots[--gc_root_count];
            return;
        }
    }
}
```

8.2 Mark Phase

```
void gc_mark(struct heap_t *h) {
    struct chunk_t *chunk = h->start;

    while (chunk) {
        chunk->marked = 0;
        chunk = chunk->next;
    }

    for (uint32_t i = 0; i < gc_root_count; i++) {
        chunk = (struct chunk_t *)(
            (uint8_t *)gc_roots[i] - sizeof(struct chunk_t)
        );
        chunk->marked = 1;
    }
}
```

8.3 Sweep Phase

```
void gc_sweep(struct heap_t *h) {
    struct chunk_t *chunk = h->start;
    while (chunk) {
        struct chunk_t *next = chunk->next;
        if (chunk->inuse && !chunk->marked) {
            hfree(h, (uint8_t *)chunk + sizeof(struct chunk_t));
        }
        chunk = next;
    }
}
```

8.4 Garbage Collection Execution

```
void gc_collect(struct heap_t *h) {
    gc_mark(h);
    gc_sweep(h);
}
```

The main function validates:

- Heap initialization
- Allocation and deallocation
- Fragmentation handling
- Garbage collection of unreachable memory
- Cleanup of heap resources

---

10. Limitations and Future Work

- Manual root registration required
- No conservative pointer scanning
- No alignment enforcement
- Not thread-safe
- Simplistic heap spraying detection

---

11. Conclusion

This project successfully demonstrates a complete custom heap allocator with safety checks, fragmentation control, and garbage collection. It provides a strong foundation for understanding real-world memory allocators and highlights common challenges in low-level memory management.