Memory Subsystem

The Memory Subsystem provides a centralized system for tracking and managing memory allocations throughout the engine. Its purpose is to offer granular control over how memory is used, help instrument performance, and assist in detecting issues like memory leaks.

By abstracting platform-specific memory functions, it allows for the future integration of custom allocators for different types of data, such as textures, arrays, or game objects. This helps avoid frequent dynamic allocations on the fly, which can be a performance bottleneck. All memory operations are funneled through this system, providing a single point of control and reporting.

Core Concepts: Tagged Allocations

A key feature of the memory subsystem is the ability to **tag** every allocation with a specific category. This is achieved through the memory_tag enumeration, which provides a list of predefined categories for common engine and game components.

This tagging mechanism allows the engine to track exactly how much memory is being used by each subsystem. For example, it can report the total memory consumed by textures, scene data, or strings.

The memory_tag Enum

The memory_tag enum defines all possible allocation categories. It includes a default MEMORY_TAG_UNKOWN for temporary or unclassified allocations and ends with MEMORY_TAG_MAX_TAGS to simplify iterating through all categories for reporting purposes.

C

```
typedef enum memory_tag {
    // For temporary use. Should be assigned a proper tag.
    MEMORY_TAG_UNKOWN,
    MEMORY_TAG_ARRAY,
    MEMORY_TAG_DARRAY,
    MEMORY_TAG_DICT,
    MEMORY_TAG_RING_QUEUE,
    MEMORY_TAG_BST,
    MEMORY_TAG_STRING,
    MEMORY_TAG_APPLICATION,
    MEMORY_TAG_JOB,
    MEMORY_TAG_TEXTURE,
    MEMORY_TAG_MATERIAL_INSTANCE,
    MEMORY_TAG_RENDERER,
    MEMORY_TAG_GAME,
    MEMORY_TAG_TRANSFORM,
    MEMORY_TAG_ENTITY,
    MEMORY_TAG_ENTITY_NODE,
    MEMORY_TAG_SCENE,

    MEMORY_TAG_MAX_TAGS
} memory_tag;
```

Note: Using MEMORY_TAG_UNKOWN is valid but will trigger a warning in the console. This serves as a reminder to properly classify all long-term allocations.

API and Usage

The memory subsystem is accessed through a clean and simple API defined in kmemory.h. These functions wrap the underlying platform calls, adding tracking and management logic.

Lifecycle

- **void initialize_memory()** Initializes the memory subsystem, primarily by zeroing out the internal statistics counters. It should be called once at application startup.
- **void shutdown_memory()** Reserved for future cleanup tasks. It should be called once before the application exits.

Allocation and Deallocation

- **void* kallocate(u64 size, memory_tag tag)** Allocates a block of memory of a given size. Unlike malloc, it requires a memory_tag and automatically zeroes out the allocated block to prevent uninitialized data.
- **void kfree(void* block, u64 size, memory_tag tag)** Frees a previously allocated block of memory. It requires the original size and tag to correctly update the internal memory usage statistics.

Utility Functions

- **void* kzero_memory(void* block, u64 size)** A passthrough to the platform's memset equivalent, filling a block of memory with zeroes.
- **void* kcopy_memory(void* dest, const void* src, u64 size)** A passthrough to the platform's memcpy equivalent, copying a block of memory from a source to a destination.
- **void* kset_memory(void* dest, i32 value, u64 size)** A passthrough to the platform's memset equivalent, filling a block of memory with a specific value.

Implementation Details

The internal workings of the memory subsystem are kept private within kmemory.c. Its primary responsibility is to track allocations and serve as an intermediary to the platform layer.

Statistics Tracking

To track memory usage, a static memory_stats structure is used. This structure holds the total allocated memory across the entire application and an array that tracks allocations for each individual memory tag.

C

```
struct memory_stats {
    u64 total_allocated;
    u64 tagged_allocations[MEMORY_TAG_MAX_TAGS];
};

static struct memory_stats stats;
```

Allocation Flow

When kallocate is called, it performs the following steps:

1. **Increments Counters:** It adds the requested size to both stats.total_allocated and the specific entry in stats.tagged_allocations corresponding to the provided tag.
2. **Platform Call:** It calls platform_allocate to get the actual memory block from the operating system.
3. **Zero Memory:** It immediately calls platform_zero_memory on the new block to ensure it is clean.
4. **Return Block:** It returns the pointer to the zeroed memory block.

The kfree function performs the reverse operation, subtracting the size from the counters before calling platform_free.

C

```
// Simplified example from kallocate
void* kallocate(u64 size, memory_tag tag) {
    // Update total and tagged statistics
    stats.total_allocated += size;
    stats.tagged_allocations[tag] += size;

    // TODO: Memory alignment
    void* block = platform_allocate(size, FALSE);
    platform_zero_memory(block, size);
    return block;
}
```

Memory Usage Reporting

For debugging and profiling, the memory subsystem provides a function to generate a formatted, human-readable report of current memory usage.

get_memory_usage_str()

This function generates a string that lists the memory consumed by each tag. It automatically scales the byte counts into more readable units: gibibytes (GiB), mebibytes (MiB), kibibytes (KiB), or bytes (B). These units are based on powers of 1024 for technical accuracy.

To provide clear labels, the function uses a static array of strings that corresponds to the memory_tag enum.

C

```c
static const char* memory_tag_strings[MEMORY_TAG_MAX_TAGS] = {
    "UNKNOWN    ",
    "ARRAY      ",
    "DARRAY     ",
    "DICT       ",
    // ... and so on for all tags
};
```

Example Output

Calling this function and printing its output to the console produces a report similar to the following:

Shell

```shell
System memory use (tagged):
  UNKNOWN    : 0.00B
  ARRAY      : 16.00KiB
  STRING     : 4.20KiB
  TEXTURE    : 12.50MiB
  RENDERER   : 2.13MiB
  GAME       : 728.00B
  // ... etc.
```

⚠ **Important**: The string returned by get_memory_usage_str() is dynamically allocated. The caller is responsible for freeing this memory after use to avoid a memory leak. However, as it is a debug tool, this is often only called in specific, controlled scenarios.