mCertikOS Function Dependencies

Memory Management

1. **`container_alloc(size_t size)`:**
    - **Usage:**
        - `message.c`: Used in `message_create` to allocate memory for the `message_t` structure.
        - `topic.c`: Used in `topic_subscribe` to allocate memory for the `subscriber_t` structure.
        - `queue.c` (or equivalent): Used within the queue implementation (e.g., in `queue_create` or when allocating queue nodes) to allocate memory for the message queue's internal data structures.
    - **Purpose:** Allocates a block of memory of the specified `size` from the current process's container. This ensures that memory usage is tracked and limited according to mCertikOS's resource quotas, preventing a process from consuming excessive memory.
    - **Importance:** *Critical*. Using `container_alloc` is essential for adhering to mCertikOS's memory management policy and preventing memory exhaustion.
2. **`container_free(void *ptr)`:**
    - **Usage:**
        - `message.c`: Used when dequeue the oldest message when the queue is full, or destroy the message if enqueue fails, or when the message is delivered successfully inside trap_handler.
        - `topic.c`: Used in `topic_unsubscribe` to free the memory allocated for the `subscriber_t` structure and message queue.

        - `queue.c` (or equivalent): Used within the queue implementation (e.g., in `queue_destroy` or when dequeuing messages) to free the memory associated with queue nodes.
    - **Purpose:** Releases a previously allocated memory block pointed to by `ptr` back to the container.
    - **Importance:** *Critical*. Failing to `container_free` allocated memory will lead to memory leaks, eventually causing the system to run out of memory.

Process Management

1. **`current_process_id()`:**
    - **Usage:**
        - `topic.c`: Used in `topic_subscribe` to get the process ID of the subscriber and store it in the `subscriber_t` structure.
    - **Purpose:** Returns the unique identifier (process ID) of the currently executing process.
    - **Importance:** *Critical*. This is necessary to identify the subscriber process so that traps can be directed to the correct process context for callback execution.
2. **Potentially `thread_create()` (If Asynchronous Callbacks are Used):**
    - **Usage:** *Potentially* used within the trap handler or a separate message delivery mechanism if callbacks are executed in separate threads to avoid blocking. This isn't strictly required for the *simplified* version if callbacks are executed directly within the trap handler (though this is less desirable).
    - **Purpose:** Creates a new thread within the current process.
    - **Importance:** *Optional for the simplified version, but highly recommended for a robust implementation*. Asynchronous callbacks prevent the publisher from being blocked by long-running or faulty callback functions.

Trap Handling

1. **`trap_set_handler(int trap_number, void (*handler)(int))`:**
    - **Usage:**
        - `trap.c`: Used in `trap_init()` to register the `trap_handler` function with the mCertikOS kernel for a specific trap number.
    - **Purpose:** Registers a function (`handler`) to be executed when a specific trap (`trap_number`) occurs. This allows the kernel to redirect control to the appropriate handler when a trap is triggered.
    - **Importance:** *Critical*. This is the core function that connects the trap mechanism to the message delivery process.
2. **`trap_send(int pid)`:**
    - **Usage:**
        - `syscall.c`: Used in `sys_pub` to trigger a trap to the subscriber process (identified by `pid`) after a message has been enqueued.
    - **Purpose:** Sends a trap signal to the process with the specified process ID (`pid`). This initiates the execution of the registered trap handler in the context of the target process. The specific implementation will vary, but this will likely involve using a system call or kernel-level function to signal the process.
    - **Importance:** *Critical*. This function is responsible for initiating the message delivery to subscribers.

Synchronization Primitives

1. **`spinlock_init(spinlock_t *lock)`:**
    - **Usage:**
        - `topic.c`: Used in `topic_init` to initialize the spinlock `goodbye_topic.lock`, also used inside `topic_subscribe` to initialize the spinlock `subscriber->lock`.
    - **Purpose:** Initializes a spinlock.

- **Importance:** *Critical*. Initializes the lock before use.

2. **`spinlock_lock(spinlock_t *lock)`:**
   - **Usage:**
     - `topic.c`: Used in `topic_subscribe` before modifying the global `goodbye_topic` data, used in `topic_unsubscribe` before unsubscribing the current subscriber, used in `message_enqueue` before enqueueing the message into the message queue.
     - `syscall.c`: Used in `sys_pub` before sending trap signal
   - **Purpose:** Acquires a spinlock, waiting (spinning) until the lock is free.
   - **Importance:** *Critical*. Prevents race conditions and ensures data consistency when multiple processes or threads access shared data structures.
3. **`spinlock_unlock(spinlock_t *lock)`:**
   - **Usage:**
     - `topic.c`: Used in `topic_subscribe` after modifying the global `goodbye_topic` data, used in `topic_unsubscribe` after unsubscribing the current subscriber, used in `message_enqueue` after enqueueing the message into the message queue.
     - `syscall.c`: Used in `sys_pub` after sending trap signal
   - **Purpose:** Releases a spinlock, allowing other processes or threads to acquire it.
   - **Importance:** *Critical*. Releasing the lock is essential to allow other processes to make progress. Failing to release a lock will lead to deadlocks.