Simplified Pub/Sub IPC Implementation

Implementing a basic publish-subscribe inter-process communication (IPC) mechanism within mCertikOS, allowing processes to publish messages to a pre-defined topic ("goodbye_topic") and have subscriber processes receive and display these messages via a callback function.

Phase 1: Foundation - Core Data Structures & Memory Management

Define core data structures for topic, subscriber, and messages, and establish memory management using mCertikOS containers.

- **Define Data Structures (pubsub.h):**
  - `message_t`: Struct containing `char data[MESSAGE_SIZE]`.
  - `subscriber_t`: Struct containing:
    - `int pid`: Process ID of the subscriber.
    - `message_callback_t callback`: Function pointer to the callback.
    - `queue_t* message_queue`: Message queue for the subscriber.
    - `spinlock_t lock`: Lock for queue access.
  - `topic_t`: Struct containing:
    - `char name`: Topic name ("goodbye_topic").
    - `subscriber_t *subscriber`: Pointer to a single subscriber.
    - `bool subscriber_present`: Flag if the subscriber is present.
    - `spinlock_t lock`: Lock for subscriber management.
  - `queue_t`: Define `queue_t` structure (consider a circular buffer).
  - Define constants: `TOPIC_NAME`, `MESSAGE_SIZE`, `QUEUE_SIZE`.
- **Implement Message Management (message.c):**
  - `message_create(const char *message_data)`: Allocates memory for a `message_t` using `container_alloc` and copies `message_data`. Returns `message_t*`.
  - `message_enqueue(subscriber_t *subscriber, message_t *message)`: Enqueues a message to the subscriber's queue, manages queue overflow (dequeue oldest message).
  - `message_dequeue(subscriber_t *subscriber)`: Dequeues a message from the subscriber's queue.
  - `queue_create(size_t queue_size)`: Creates a message queue of size `queue_size`, using `container_alloc` for internal storage.

    - `queue_destroy(queue_t* queue)`: Destroys a message queue, freeing allocated memory with `container_free`.
    - `queue_enqueue(queue_t* queue, message_t* message)`: Enqueues a message to the queue.
    - `queue_dequeue(queue_t* queue)`: Dequeues a message from the queue.
    - `queue_is_full(queue_t* queue)`: Checks if the queue is full.
- **Implement Topic Initialization (topic.c):**
    - `topic_init()`: Initializes the global `goodbye_topic` structure with the topic name, setting `subscriber` to `NULL` and `subscriber_present` to `false`, initializing `spinlock`.

Phase 2: Topic and Subscriber Management

Implement functions for subscribing and unsubscribing to the pre-defined topic.

- **Implement `topic_subscribe` (topic.c):**
    - Checks if `topic_name` is equal to `TOPIC_NAME`.
    - Acquire `goodbye_topic.lock`
    - Checks if a subscriber already exists (`subscriber_present`). If so, return an error.
    - Allocates memory for a `subscriber_t` using `container_alloc`.
    - Allocates memory for `message_queue` using `queue_create`.
    - Sets `subscriber->pid` to the current process ID (`current_process_id()`).
    - Sets `subscriber->callback` to the provided callback function.
    - Initializes `subscriber->lock`.
    - Set `subscriber_present` to true.
    - Releases `goodbye_topic.lock`.
    - Returns success or failure code.
- **Implement `topic_unsubscribe` (topic.c):**
    - Checks if `topic_name` is equal to `TOPIC_NAME`.
    - Acquire `goodbye_topic.lock`
    - Checks if a subscriber exists (`subscriber_present`). If not, return an error.
    - Frees the `message_queue` using `queue_destroy`.
    - Free the `subscriber_t` structure using `container_free`.
    - Sets `subscriber_present` to `false`.
    - Releases `goodbye_topic.lock`.
    - Returns success or failure code.

Phase 3: System Call Integration

Expose the Pub/Sub functionality through system calls.

- **Add Syscall Definitions (syscall.h):**
  - Define system call numbers for `SYS_PUB`, `SYS_SUB`, and `SYS_UNSUB`.
- **Implement Syscall Handlers (syscall.c):**
  - `sys_sub(const char *topic_name, message_callback_t callback, size_t queue_size)`: Calls `topic_subscribe` with the provided arguments.
  - `sys_unsub(const char *topic_name)`: Calls `topic_unsubscribe` with the provided `topic_name`.
  - `sys_pub(const char *topic_name, char *message_data)`:
    - Checks if `topic_name` is equal to `TOPIC_NAME`.
    - Acquires `goodbye_topic.lock`.
    - Checks if a subscriber exists. If not, return an error.
    - Creates a `message_t` using `message_create`.
    - Enqueues the message using `message_enqueue`.
    - Releases `goodbye_topic.lock`.
    - Calls `trap_send(subscriber->pid)` to trigger the callback execution in the subscriber process.
    - Returns success or failure code.
- **Integrate with mCertikOS Syscall Mechanism:**
  - Modify the mCertikOS kernel to handle the new system calls (add entries in the syscall table, etc.).

Phase 4: Message Delivery and Callback Execution

Implement the trap-based mechanism for delivering messages and executing callbacks in the subscriber process.

- **Implement `trap_send(int pid)` (trap.c):**
  - This function needs to trigger a trap to the process with an ID `pid`. This will likely involve using mCertikOS-specific functions to send a signal or interrupt to the target process.
- **Implement `trap_handler(int trap_number)` (trap.c):**

    - This function is the trap handler that will be executed in the *subscriber's* context.
    - Verify that the trap is for the correct reason/number.
    - Acquire `goodbye_topic.lock`.
    - Check if a subscriber exists and that the PID matches the current process.
    - Dequeues the message using `message_dequeue`.
    - Releases `goodbye_topic.lock`.
    - If a message was dequeued:
        - Calls the subscriber's callback function (`subscriber->callback(message->data)`).
        - Free the message using `container_free`.
- **Implement `trap_init()` (trap.c):**
    - Initialize the trap handler by calling `trap_set_handler` (or equivalent mCertikOS function) to register the `trap_handler` function for a specific trap number.
- **Modify `subscriber.c`:**
    - Call `trap_init()` after subscribing to the topic to initialize the trap handler.

Technical Challenges and Mitigations:

- **Challenge 1: Synchronization:**
    - Solution: Use spinlocks carefully to protect shared data structures. Avoid holding locks for extended periods.
- **Challenge 2: Trap Handling Complexity:**
    - Solution: Carefully study the mCertikOS trap mechanism and ensure that the trap handler is correctly implemented and executed in the subscriber's context.
- **Challenge 3: Memory Management:**
    - Solution: Use `container_alloc` and `container_free` consistently to adhere to mCertikOS's resource quotas. Check return values from allocation functions and handle failures gracefully.
- **Challenge 4: Deadlock:**
    - Solution: Be careful when using spinlocks to avoid deadlock, avoid nested locks.