

Introduction

In this assignment you will complete an implementation of a `message queue` by adding the necessary `locks` and `condition variable` operations to synchronize access to the message queue. You will also implement I/O multiplexing functionality - `monitoring multiple queues for events` (e.g. new messages) in a single thread - modeled after the `poll()` system call.

Part 1: Message queue implementation

A message queue is very similar to a pipe. The key difference is that a pipe is a stream of bytes, and a message queue stores distinct messages by first storing the message length (as a `size_t` type), and then the message itself. This means that a receiver retrieves a message by first reading the `size` of the message, and then reading `size` bytes from the message queue.

Your first task is to complete the functions in `msg_queue.c` with the exception of `msg_queue_poll()`. The description of each function is in `msg_queue.h`. Plan to spend some time studying these two files so that you understand the underlying implementation of the message queue.

For example, most of the `msg_queue_create()`, `msg_queue_open()`, and `msg_queue_close()` functions are given to you. Notice that `msg_queue_create()` allocates space for and initializes a data structure (`mq_backend`) that maintains all the state for the queue. `You will need to add your synchronization variables here`. Also, notice that `msg_queue_create()` returns a message queue handle (much like a file descriptor). Trace through the code to see how this handle is created and returned.


The starter code also includes a ring buffer (`aka circular buffer`) to store the messages. The file `ring_buffer.h` describes the functions used to read and write from the buffer as well as to check how much space is used or free.

Part 2: Implementing `msg_queue_poll()`

The task of `msg_queue_poll()` is to wait until one of a set of multiple queues becomes ready for I/O (read or write). `The API that you need to implement is modeled after the poll() system call, so please read the manual page for it (man 2 poll)`. Note that you do not need to implement the timeout feature of the Linux `poll`. We will provide a simple test program that uses `msg_queue_poll()`.

Your `msg_queue_poll()` function should consist of 3 stages:

1. Subscribe to requested events. `msg_queue_poll` takes an array of `msg_queue_pollfd`. This array contains information about the message queues that poll call will be monitoring including the events that a thread has requested.
2. Check if any of the requested events on any of the queues have already been triggered: If none of the requested events are already triggered, block until another thread triggers an event. Blocking should be implemented by waiting on a condition variable (which will be signaled by the thread that triggers the event). Note that you will need to keep track of whether an event has been signalled and which thread to wake up when an event is signalled.
3. After the blocking wait is complete, determine what events have been triggered, and return the number of message queues with any triggered events. Like poll and select, the caller will then check each message queue it is subscribed to for an event.

See the [multiprod.c](#)  example for how `msg_queue_poll` is used. (Here is an updated [Makefile](#) that will also compile the new program.) This program also measures how much time is spent in read and poll calls. We would expect to see the time spent in poll to be much higher than read, if we poll was implemented correctly. However, to see this effect you need to use the NDEBUB flag when compiling to remove delays and overhead of the mutex validators.

The starter code contains a doubly-linked list implementation (file `list.h`) that you should use for the wait queue.

Important Details

We have implemented some wrappers and synchronization elements to facilitate testing your code, so it is important that you follow the instructions in the starter code with respect to which files you are allowed to change.

>You must use the synchronization functions provided in `sync.c`. These are wrappers around the pthread versions of the synchronization functions, and we may instrument these wrappers to generate extra output for testing.

The `mutex_validator` is a technique used to test whether critical sections are being accessed by more than one thread at a time. You can see calls to the validator in the ring buffer code and in the linked list code. Note that the validator locks only protect the validator data structure they are not involved in the synchronization of access to the queue data structures.

You will need to use the linked list implementation provided in `list.h` for your wait queue. It uses the same approach as the linked list implementation found in the Linux kernel. You may want to look at a [tutorial \(Links to an external site.\)](#) of how the embedded linked list code works.

What to Submit

You know the drill by now. Please make sure to commit all of the files needed to compile and run your program. Pay attention to which files you are allowed to change. We do not recommend adding additional source files containing helper functions, so check with the instructor if you think you really need to. You are welcome to add additional tests programs. Please remember not to commit executable files or object files.