Assignment 2: Message Queues

Due Nov 15, 2021 by 10p.m. **Points** 10

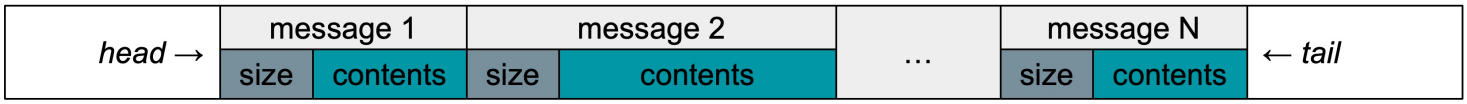
Starter code is available on MarkUs

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.

You will not be making any changes to `ring_buffer.c` or `ring_buffer.h,` and will only interact with the ring_buffer via the ring_buffer functions.

Complete and commit your work on this part of the assignment before moving on to implement poll. The `prodcon.c` program can be used to test it.

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 and 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 signaled and which thread to wake up when an event is signaled.
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. 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 `NDEBUG` 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 will use for the wait queue. There is **video tutorial (https://web.microsoftstream.com/video/a234acf2-cd1d-41b7-813d-e68ca796b606)** to help you understand how the linked list code works, and an

example program, **ll_example.c** (https://q.utoronto.ca/courses/234190/files/17283666/download?download_frd=1) that uses a slightly modified **list.h.** (https://q.utoronto.ca/courses/234190/files/17283667/download?download_frd=1) (The list.h file for the example does not include all of the additional validator code.)

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. NOTE: the mutex validator is disabled in non-debug build (if the `NDEBUG` flag is defined)

You will need to used 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** **(https://medium.com/@414apache/kernel-data-structures-linkedlist-b13e4f8de4bf)** of how the embedded linked list code works.

Your code must compile cleanly both with the `NDEBUG` flag defined and undefined.

Various notes

- Most of the error conditions that you need to check for can only happen if the API is not used correctly by a program using this library (i.e. programmers' mistakes). This is very similar to how a syscall implementation has to check the validity of the arguments, taking into account the current state of the system.
- Use `report_error()` and `report_info()` for error output (defined in `errors.h` in the starter code; see the comments in this file for details).
- Queue operations must preserve the message boundaries and atomicity of reads and writes. The `ring_buffer_peek()` function is useful for reading message length without removing it from the queue buffer.

- `msg_queue_poll()` reports that a queue is ready for I/O "right now" - at a moment shortly before the call returns. By the time the program can act on this information with a read or write operation, the state might have already been changed by a different thread. This is the main reason why we still need non-blocking reads and writes: a blocking operation after a poll call might still block indefinitely (e.g. if another thread reads the message instead).
- The condition variable to wait on in `msg_queue_poll()` can be created specifically for this call.
- You will need to keep track of the threads currently blocked in a `msg_queue_poll()` call for a specific queue, so that they can be notified when necessary. This can be done by storing a list of corresponding condition variables ("wait queue").
- One of the most common mistakes in the poll implementation is a race condition where a condition variable signal is "missed" by the thread waiting for it. You might find the `COND_AUTO_WAKEUP` debugging option (see `sync.c` in the starter code) useful in identifying and debugging this issue.
- The helgrind tool (part of valgrind) could be helpful in debugging synchronization issues.