CSC369 Tutorial:
Intro to Assignment 2

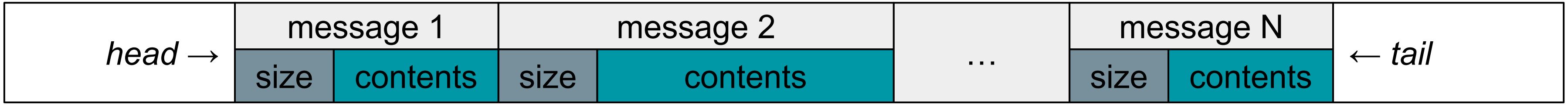
Alexey Khrabrov, CSC369 Fall 2020, University of Toronto

A2 overview

- Goal: implement a message queue
    - Similar to a pipe; main difference - message boundaries
    - Main focus on synchronization
- Part 1: read/write operations
    - Classic producer-consumer plus *close* operations and *non-blocking* reads/writes
- Part 2: poll - I/O multiplexing (waiting until at least one queue is ready)
    - Modeled after the `poll()` system call
    - The hard part of the assignment
- Starter code fully specifies the API and provides an implementation skeleton
- Much “smaller” assignment compared to A1
    - Our solution adds < 300 LOC to starter code
- Difficulty stems from complexities of synchronization

Message queue

- Underlying storage - ring buffer of given capacity



head →

message 1

size

contents

message 2

size

contents

…

message N

size

contents

← *tail*

- Message consists of size and contents
  - Use `size_t` to store the size
  - Contents - blob of `size` bytes
    - **Not** a null-terminated string (in general)
- Read/write operations are *atomic* and preserve message boundaries
  - Read and write one whole message (including size header) at a time
  - Write: if not enough free space for the whole message - block until enough space
  - Read: if next message is larger than user-supplied buffer - return error, leave the queue as is
    - Need to read the size without extracting it from the ring buffer

Queue handles

- API refers to a queue by a *handle*
  - Opaque identifier - similar to a file descriptor
  - `msg_queue_t msg_queue_create(size_t capacity, int flags);`
- Can open additional handles to an existing queue
  - Similar to duplicating a file descriptor
  - `msg_queue_t msg_queue_open(msg_queue_t queue, int flags);`
- Handle = reference to queue + *flags*
  - Determine what operations are allowed for this handle and their behaviour
- *Reader* handles and *writer* handles
  - `MSG_QUEUE_READER` and `MSG_QUEUE_WRITER` flags
- *Blocking* (default) and *non-blocking* handles
  - `MSG_QUEUE_NONBLOCK` flag

Close semantics

- Queue is destroyed when all handles are closed
    - Reference count reaches 0
- What if all *reader* handles are closed, but there are still *writer* handles left
    - Or the other way around
    - Similar to closing only one end of a pipe
    - NOTE: *newly created* queue with no reader (or writer) handles *yet* - different case
- If we do nothing, already blocked writers (or readers) would wait forever
- Need to notify readers when closing the last writer handle
    - “End of file” condition - like closing the write end of a pipe
- Need to notify writers when closing the last reader handle
    - “Broken pipe” condition - like closing the read end of a pipe
- “Other side” will wake up and fail the operation

Non-blocking operations

- Similar to e.g. non-blocking socket send/recv operations
- Reads/writes on handles open with the `MSG_QUEUE_NONBLOCK` flag
- Non-blocking write: fail with `EAGAIN` if not enough free space
- Non-blocking read: fail with `EAGAIN` if the queue is empty

- NOTE: do **not** implement blocking ops as a busy loop, use condition variables writer/producer

```
lock()

while(buffer is full){
    wait(&empty);
    if there is no reader > exit()
}

Do the write op


signal(&full);

unlock()
```

reader/consumer

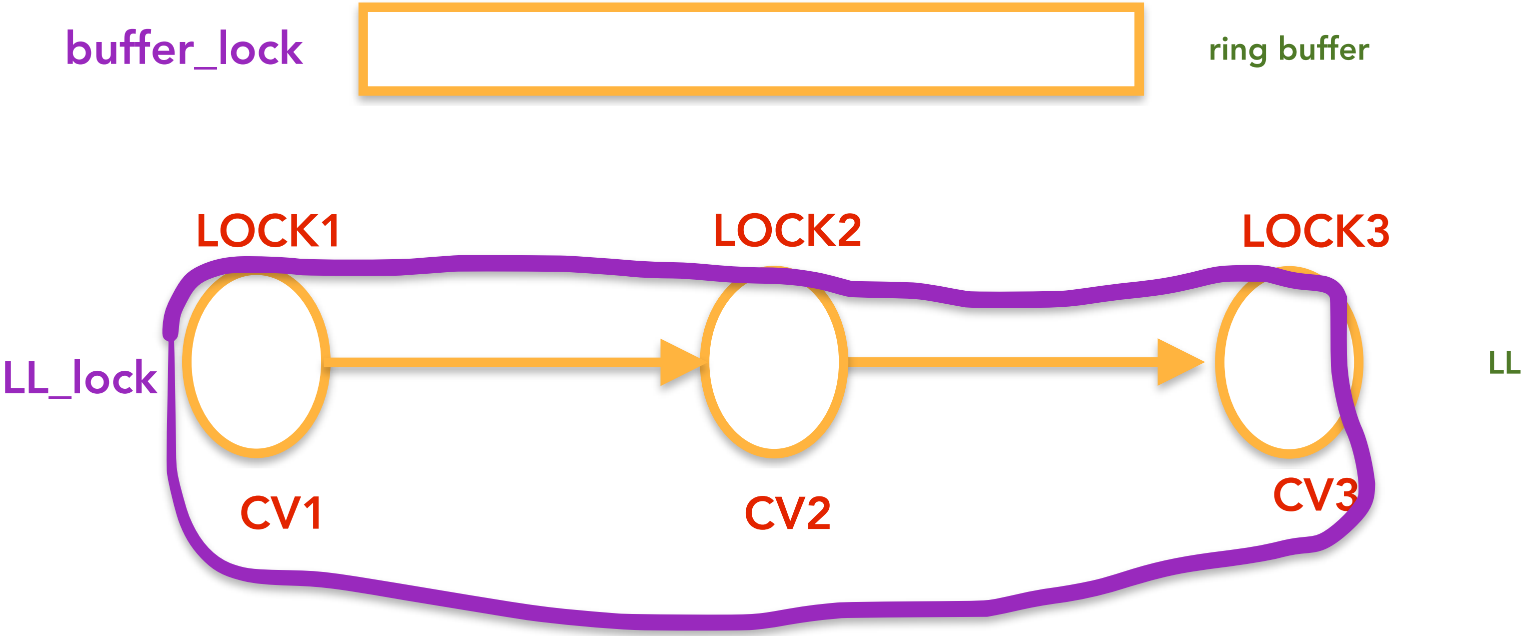
```
lock()

while(buffer is empty){
    wait(&full);
    if there is no writer > exit()
}

Do the read op

signal(&empty);

unlock()
```



buffer_lock

ring buffer

LOCK1

LOCK2

LOCK3

LL_lock

LL

CV1

CV2

CV3

Poll

- Purpose - wait until at least one queue in a set is ready (I/O multiplexing)
- Data structures
  - Queue: keep track of the threads currently blocked in poll calls for this queue
  - Polling thread: keep track of the current state of the queues it's waiting for to become ready
- Implementation outline
  - 1. Subscribe to events (so that other threads know when to notify this thread)
  - 2. Check if any events already triggered, otherwise block
  - 3. Check what events were triggered, return info to caller
- Note: reports what queues are ready *right now* - shortly before call returns
  - State can change by the time thread calls read or write
- Hints
  - Condition variable to wait on can be created dynamically for this poll call
  - Can use a linked list for the "wait queue" - threads waiting for a queue to become ready

Starter code overview

- msg_queue.h - API definitions and documentation/specification
- msg_queue.c - API implementation
  - NOTE: this is the **only** file you will modify
- ring_buffer{.h, .c} - underlying ring buffer implementation
- sync{.h, .c} - synchronization primitives
  - Wrappers around pthread functions - allow us to intercept them for testing purposes
- list.h - doubly-linked list implementation (useful for poll wait queue)
- errors{.h, .c} - error logging helpers
- mutex_validator{.h, .c} - synchronization debugging/testing tool
- prodcon.c, multiprod.c - example programs/tests
- Makefile - the only thing to change is compile flags (during development)

Linked list

- Can be (and should be) used to implement the wait queue for poll
- “Embedded” entries - different from linked lists you are used to

  -
    ```
    struct s {
        ...data fields...
        list_entry entry;// stores next and prev pointers
    };
    ```
- To get pointer to containing struct from pointer to entry:
  - `struct_ptr = container_of(entry_ptr, s, entry);`
- More efficient than having separate struct with next, prev, data fields
  - Not wasting memory on storing data pointers
  - Avoiding lots of small dynamic memory allocations

Synchronization debugging tools

- Mutex validator
  - Checks that operations that must be done inside a critical section are actually done by one thread at a time
  - Adds delays to extend duration of critical sections to make violations more likely
  - Included in ring buffer and linked list
  - You can use it in your data structures for implementing poll
- Early wakeups from condition variable waits
  - Can be useful in debugging cases when waiting thread “misses” a signal and waits forever
  - See details in sync.c
- External tools
  - Helgrind (part of Valgrind) - able to detect various types of synchronization error

What to implement in msg_queue.c

- Part 1
  - `struct mq_backend`, `mq_init()`, `mq_destroy()`
    - Synchronization variables and their initialization
  - `msg_queue_write()`
  - `msg_queue_read()`
  - `msg_queue_close()`
    - Synchronization and notifying the other end if closing the last reader or writer handle
- Part 2
  - `struct mq_backend`, `mq_init()`, `mq_destroy()`
    - Data structures to keep track of threads blocked in poll for this queue
  - `msg_queue_poll()`
  - `msg_queue_close()`
    - Notifying polling threads if closing the last reader or writer handle

Important notes

- Read the handout and doc comments in starter code carefully a few times
- Common mistakes
    - Not preserving atomicity of reads and writes
        - Use `ring_buffer_peek()` to read message size
    - Incorrect use of condition variables; should always do this:

      ```
      while (!predicate) {
          ...check if still need to wait...
          cond_wait(&cond, &mutex);
      }
      ```
    - Thread waiting forever in poll after missing a condition variable signal
- Most of error checking is about API misuse - programmers’ mistakes
- Use macros from errors.h in the starter code for error logging