1. Introduction

Writing correct concurrent programs using shared data and locks is hard.

To use this model, one decides what data will be shared by multiple threads and protects access to that data with locks, ensuring mutual exclusivity. There are a multitude of difficulties in this approach. Programmers find this model to be difficult to reason about, leading to hard-to-find bugs. Testing can be unreliable; code may be tested thousands of times over but never trigger a clandestine bug. When bugs do appear, programmers will often solve them by adding locks, running the risk of over-synchronization, which can impact performance. Even worse, a significant portion of concurrency bugs are caused by violation to programmers’ order intentions [1] which cannot be solved by simply locking.

Fortunately, an alternative strategy exists that is suitable for a wide range of concurrent programs. Actor-based concurrency is a share-nothing approach in which 'actors' send and receive 'messages' rather than reading and writing to shared memory.

This paper contains the following:

- A presentation of simple C++ Actor-based concurrency API suitable for writing arbitrary concurrent programs.
- Two solutions to the producer-consumer problem, one using the Actor API, one using threads and a shared buffer.
- A comparison of the difficulty in reasoning about the two solutions.
- A comparative benchmark of the two solutions.

The producer-consumer problem is an ideal locus for highlighting some of the pitfalls of the shared-memory model of multi-core programming and conversely the advantages of Actor-based concurrency; easier to conceptualize, no state-sharing to cause unanticipated bugs, and competitive performance.

Actors address the aforementioned problems with shared data and locks by providing an approach that is easier to reason about and is competitively performant. This paper provides an overview of the basic components of an actor system through the example implementation of the producer-consumer problem.

2. Actor Fundamentals

The main idea with actor-based concurrency is that instead of using shared memory, communication is modeled with actors. Actors are concurrent, thread-like entities that communicate by sending messages to each other. Synchronization is achieved because a message can be received only after it has been sent. [2] Although thread-like, an actor may or may not have a 1:1 relationship with a thread. Actors are typically scheduled over a thread-pool.

An actor has a mailbox for receiving its messages. When an actor sends a message to a mailbox, it does not block, when an actor receives a message in its mailbox, it is not interrupted. An actor's mailbox can be thought of as its work queue. Actors usually have an 'act' method which contains a loop. The loop iterates reading messages from its mailbox and doing work in response.
A simple example is a thread-safe counter. In an actor-based system, one would model the counter as an actor. Any entity could send the counter actor 'increment' messages. At any time, any entity could send the counter a 'total request' message; the actor would respond with its running total at that time.

```cpp
void Counter::act() {
    int total = 0;
    while (true) {
        Message* m = Actor::receive();
        if (m->type() == "INC") {
            total++;
            delete m;
        } else if (m->type() == "TOTAL") {
            m->getSender()->Actor::send(new Total(this, total));
            delete m;
        }
    }
}
```

Although Actor-based concurrency systems use locks and shared data internally, none of this is exposed to the programmer. The focus is on giving the programmer an interface that conceals these details and allows them to focus in on decomposing the problem at hand.

3. An Implementation

What follows is a interface definition of a custom actor library written in C++. We later use this library to show a solution to the producer-consumer problem.

3.1 Actor

**Interface:**
Actor is a virtual class to be extended and the act method implemented.

```cpp
// The actor begins reading message from its mailbox.
// numActors is the number of logical actors (threads) used initially.
void start(int numActors = NUM_CORES);

// Put a message in this actor's mailbox
void send(Message* m);

void stop();

// To be implemented by subclasses to do arbitrary work
virtual void act() = 0;

// An actor retrieves a message from its own mailbox
Message* receive(); // protected
```

**Implementation:**
The Actor class has three important members: a thread pool, thread-safe, bounded FIFO queue and a load factor buffer. At construction, for each logical actor, a callback on its act method is put into the thread pool.

At the call of start, each logical actor will start reading from the mailbox.

At each call of send, a message is placed in its queue. The load factor is stored in the load factor buffer. At the call of stop, the actor will shut down. Existing messages will be processed, but no new are accepted
Possible Improvements:
A pattern matching capability could be implemented to ease
the detection of message type. Also, we could provide optional configuration
for synchronous or asynchronous message receipt (send() could optionally block)

3.2 Mailbox

Interface:
The mailbox is a class member of Actor. It is responsible for queuing
messages bound for its owning actor.

// Inserts message into mailbox to be read by owning actor
void write(Message* m);

// Reads the oldest message from the queue
Message* read();

// Shut the mailbox down, no new messages accepted.
void stop();

Implementation:
The mailbox is the thread-safe, bounded FIFO queue. It blocks actors on
reads when empty. When its capacity is reached, sends cause it to drop the
oldest message from the queue.

Possible Improvements:
The mailbox could block senders when full.

3.3 Load Factor Buffer

Interface:
The load factor buffer is a class member of Actor. It is responsible
for tracking and reporting the load factor of the message queue. It is not
thread-safe, simply because some level of inaccuracy is acceptable in favor
of performance.

// Records load factor at a moment in time. Called at each message receipt.
void write(double);

// Reports the load factor of the mailbox over the last 1000 receipts.
double avgLoad();

Implementation:
The load factor buffer is circular buffer.

Possible Improvements:
Could asynchronously monitor the health of the actor from a another thread.
Could be made a lock free data structure to achieve performance and accuracy.

3.4 Message

Interface:
The message is a virtual class which is to be extended for specific actors.

// Constructs a message with a pointer to its sender
explicit Message(Actor* sender) {}

// Must be implemented by subclasses
virtual std::string type() = 0;
Implementation:
A message is a virtual class. Message type is defined simply by a string.

Possible Improvements:
Ideally this class would not be necessary at all, and any type would work as a message. Also, it is expected that the sender be another Actor. A message should be able to come from any sender.

3.4 Executor

Interface:
The Executor can be thought of as the scheduler for a group of actors. After the construction of the needed actors, the user registers these actors with the Executor, specifying a maximum number of threads allowed for all actors registered with the Executor instance. This number must be carefully considered by the user, taking into consideration the io-boundedness of the actors act methods.

```
explicit Executor(int maxThreads) {}
```

// Register the actor for monitoring and scheduling
```
void register(Actor a);
```

Implementation:
The Executor can see the internals of every registered actor. It monitors the load factor of the mailbox of each actor. If a particular actor is unable to keep up with the rate of input to its mailbox, the Executor may grant it more threads (logical actors). Likewise, if an actor maintains a very low load factor, the Executor may take threads away.

Possible Improvements:
In the case that maxThreads are in use by all actors, the Executor will thrash as it tries to delegate threads to the most needy at any moment. This could degrade performance dramatically. One improvement would be to introduce a priority scheme for scheduling, giving some actors priority. Based on the priority, the system could be configured to degrade gracefully. Further, this type of scenario would need to have a mechanism to warn interested humans.
4. Producer-Consumer

Producer-consumer is perhaps the classic synchronization problem. It is typically solved by using a shared buffer which is accessed concurrently by both entities. This leads to a host of complexities which are exacerbated when multiple producers and consumers are introduced.

Alternatively, modeling this problem with actors is quite straightforward and scales to arbitrary number of producers and consumers with no additional complexity.

4.1 Producer-Consumer with Shared Data

Implementing the producer-consumer problem in a shared-data paradigm is tricky, this is especially true if supporting multiple producers/consumers is a requirement. In the traditional solution to a single producer-consumer problem, a lock around a count combined with appropriate signals and waits suffices.

```c
void produce()
{
    mutex.lock();
    while (buffer.full()) {
        pthread_cond_wait(not_full, mutex);
    }
    buffer.put(item);
    pthread_cond_signal(not_empty);
    mutex.unlock();
}

void consume()
{
    mutex.lock();
    while (buffer.empty()) {
        pthread_cond_wait(not_empty, mutex);
    }
    buffer.remove(item);
    pthread_cond_signal(not_notfull);
    mutex.unlock();
}
```

Upon adding multiple producers/consumers, this approach no longer works. The initial problem is that the signaling is essentially a commitment to the signaled process that it can consume or produce the available resource. If a signal is missed, the result is either an item is produced into a full buffer or a consumer tries to consume from an empty buffer. When multiple consumers are allowed, there has to be coordination between the signaling commitment and the while condition. In other words, we have to count the commitments, not just whether the buffer is full in the case of the producer or empty in the case of the consumer. If there is one item in the buffer and the producer signals a consumer, that consumer should receive it regardless of whether or not another consumer enters at that moment.

In addition, the number of consumers and producers waiting needs to be counted to ensure that signals are not lost (remember, every signal is a commitment). The thing to note here is if only one consumer is waiting and a producer signals, then another producer signals, the second signal is lost; we should not signal more than there are threads waiting for that signal. To address this problem, we add two
counters for the number of producers and consumers waiting. Finally, there has to be coordination around termination. If consumers are waiting and all producers are done producing, we should broadcast the consumers to finish (and have a boolean escape for the consumers to exit without consuming).

The final code has 6 counters and a boolean, as well as a function to allow the producers to indicate that they are done producing. The final remove function, below, exemplifies the problem at hand: most of the code has to do with guarding against problems that are non-intuitive and frankly hard to think of. Every assumption has to be questioned and some of these race conditions may only arise rarely.

```c++
void ConsumerShared::remove() {
  mybuf_->mutex_->lock();
  bool removed = false;
  while (mybuf_->circularbuffer_.getCount() - mybuf_->reservedItems_ == 0 && !mybuf_->done) {
    mybuf_->consWaiting_++;
    mybuf_->condVar_notempty_->wait(mybuf_->mutex_);
    //if we're not done or we're done but there's an item reserved for me
    if (!mybuf_->done || (mybuf_->done && mybuf_->reservedItems_ > 0)) {
      mybuf_->reservedItems_--;
      removed = true;
    }
  }
  if (mybuf_->circularbuffer_.getCount() > 0 && !removed) {
    removed = true;
  }
  if (removed) {
    mybuf_->circularbuffer_.read();
  }
}
```

---

**Producer**

```c++
void produce() {
  mutex.lock();
  while (bufferfull()) {
    pthread_cond_wait(not_full, mutex);
  }
  bufferput(item);
  pthread_cond_signal(not_empty);
  mutex.unlock();
}
```

**Consumer**

```c++
void consume() {
  mutex.lock();
  while (bufferempty()) {
    pthread_cond_wait(not_empty, mutex);
  }
  bufferremove(item);
  pthread_cond_signal(not_notfull);
  mutex.unlock();
}
4.2 Producer-Consumer with Actors

Solving producer-consumer using actors is quite different from the shared-memory approach. Since there is no need to be concerned about synchronization, reasoning is much easier.

1) Create a consumer actor, for which \(\text{NUM\_CORES}/2\) act() method callbacks are created.
2) Create a producer actor, for which \(\text{NUM\_CORES}/2\) act() method callbacks are created.
3) The consumers send 'READY' messages to the producers.
4) The producers receive 'READY' and responds with a 'PRODUCT'.

Solution using our API:

```cpp
ConsumerActor::ConsumerActor(ProducerActor* p) : Actor(), p_(p) {
    for (int i = 0; i < numConsumers_; ++i) {
        p_->Actor::send(new Ready(this));
    }
}

void ConsumerActor::act() {
    while (true) {
        Message* m = Actor::receive();
        if (m->type() == "PRODUCT") {
            p_->Actor::send(new Ready(this));
            delete m;
        } else if (m->type() == "STOP") {
            delete m;
            break;
        }
    }
}

void ProducerActor::act() {
    while (true) {
        Message* m = Actor::receive();
        if (m->type() == "READY") {
            m->getSender()->Actor::send(new Product(this));
            delete m;
        } else if (m->type() == "STOP") {
            delete m;
            break;
        }
    }
}
```

**Diagram:**

![Diagram of Producer and Consumer Actors](image-url)
4.3 Shared State vs Actor Benchmark

As a simple benchmark, we measured the time needed to produce and consume an arbitrary number (10000) of items. We then measured this using up to 8 threads and recorded the time for both the shared-memory producer consumer and the Actor-based producer consumer. This benchmarking was performed on the NYU energon1 machine, which is an eight core, 1.8GHz Intel Xeon, 4 GB memory machine. Overall, the time needed was very similar between the two, indicating that the message passing producer consumer is competitive with the shared-memory version. The authors believe with some additional performance optimization these lines could be brought even closer together.

5. Conclusion

Reasoning with actors is far easier, bringing with it none of the concerns about synchronization that are typically associated with concurrent programming. With the underpinnings of an actor system in our basic implementation, it should be clear that actors can provide an easier-to-use abstraction of concurrency problems.

6. References

[1] Lu, Park, Seo and Zhou. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics