

This page discusses some overall system architectures and their pros and cons.

The general question is how do we make sure high priority work gets done with low enough latency while dealing with deadlock and priority inheritance while making the application developer's life as simple as possible.

There are system issues with priority inheritance and deadlock, when there are threads using shared data structures. Locks prevent high priority work from being performed.

General Issues

- Scheduling High Priority Work - If high priority work comes in while lower priority work is coming in, how do we make sure the high priority work gets handled in a bounded period of time?
- [Priority Inheritance](#)
- [Dead Lock](#)
- [OS Latency](#)
- Shared Locks - The problem with threading is shared locks between different applications.
- Ease of Programming Model
- Robustness of Programming Model
- Minimizing Lock Scope
- Flow Control
- Reasonable Use of Memory and Tasking
- [Time Slice](#)
- Starvation

Discussion

For example, a management station performs an iterator [GetNext](#) request on a table. An object create operation comes in for the same table, which has a higher priority than the [GetNext](#) request. The create is blocked on the [GetNext](#) container lock.

As serialization is occurring this lock may be held for a relatively long period of time.

A solution is for the [GetNext](#) to give up the lock every time through its loop and reestablish the cursor every time through the loop. Not something you would ever do naturally.

Interestingly, this ends up being a form of cooperative multitasking. I don't see a way not to do this in one form or another.

Old apple and windows systems originally used cooperative multitasking when they didn't have real threads. You may have seen code with yields sprinkled everywhere. The systems usually performed horribly because the scheduling was always unfair.

Threads help with latency. The object create can get scheduled immediately, which is important in a real-time system.

But when there's a shared lock anywhere we are back to potentially poor scheduling because the high priority task must wait on the low priority task. This requires anyone who shares the lock to know there is high priority work and never to take more time than the worst case performance needed for the high priority work. Of course, this changes dynamically as code changes and with different load scenarios.

So don't share a lock. That's ideal. Hopefully your app will allow you this option. Though calls like memory allocation get you into the shared lock business. Thread specific memory pools are a potential way around this.

Not sharing locks is harder than it looks because applications have multiple legitimate clients. A container access objects and objects access containers. How can they not conflict? There are non blocking locking algorithms that are very complex. There a copying strategies that can reduce locking, but we don't have a lot of memory, and any copy must eventually take a lock.

On Threading

Threads are somewhat of a religious issue. A lot of people [dislike](#) threads because multithreaded programming is difficult. This is true when it is done incorrectly. In an environment like [Erlang](#) multithreading is safe and efficient. In Java or C++ it can easily be unsafe and inefficient.

So why do we use threads at all?

- [latency](#). In a real-time system some work must be performed with very low latency. In a [run to completion](#) approach latency can not be guaranteed.
- **priority**. In a real-time system we want more important work to be performed over lower priority work. Threads provide this ability because they can be preempted.
- **fairness**. Preemption allows multiple threads to make progress on work. One thread doesn't starve everyone at the same priority.
- **policy domain**. A thread can be a domain for policies that you may or may not want applied to different applications. Examples are: priority, floating point safety, stack space, thread local storage, blocking, memory allocation.

The key to safe thread usage is not using shared data structures protected by locks. It is lock protected shared data structures that cause latency issues, deadlock, and priority inheritance. It is nearly impossible for a human to make a correct system when any thread can access shared state.

Threading can also be bad when thread scheduling is slow. On real-time systems threads are usually very efficiently scheduled, so it is not a concern.

Lock Used Within an Application is OK?

An application that uses multiple threads and has a lock usable just within the application may not, if care is taken, suffer from the same problems as when locks are arbitrarily used from different application threads.

A single application can:

- guarantee the amount of time spent blocked on the lock
- prevent deadlock because it probably won't do work in different threads
- prevent priority inheritance

Thread Model

Applications don't have their own thread, they are always operating out of someone else's thread.

Dead lock in this model is impossible to prevent as application complexity increases. You can never be sure what code will access what locks and what threads are involved in these locks. Latency can't be guaranteed because you don't know what operations are being performed in your execution path. Priority inheritance becomes a big problem.

Actor Model (1-1)

In the [Actor](#) model all work for an application is done in a single thread context. An application maps to one thread, thus it is 1-1.

This model eliminates locks which eliminates priority inheritance and deadlock issues. But we still have the latency for high priority work issues.

In an Actor context we could introduce the idea of cooperation by having a priority associated with each message. If a higher priority message comes in on the queue then a thread variable "[GiveUpWork](#)" is set. Code in a sensitive place would check this flag at particular points in its code and stop what it is doing so the higher priority work can be scheduled.

The [GetNext](#), for example, at the top of the loop would check the [GiveUpWork](#) flag and stop what it is doing. In this case the client is guaranteed a batch size so stopping at any point is ok. We could also cause the request to be put back on the queue so it could be tried later. Notice we are potentially starving lower priority work.

An implication is to change the actor's priority accordingly with the highest priority work in its queue.

This is another form cooperative multitasking. It has the advantage of eliminating deadlock and priority inheritance. But it is kind of ugly.

Actor With Thread Pool (1-N)

Like the Actor model but a thread pool is used instead of just one thread. Only shared locks under complete control of the application are required.

Actor Arbitrary Thread Model

A hybrid is an application may have an Actor so it has a thread but other threads access the actor so shared locks are necessary to protect data. It's easy to have a lot of this because not everyone is disciplined about using only message passing between actors.

This is some ways the worse of all possible worlds. The Actor model is supposed to be a safe programming model, but instead it degenerates into a thread and lock nightmare. Selecting a language that implements a pure model will prevent this sort of hybrid.

Multiple Actor Model

Work can be routed to multiple actors. Low priority work can go to one actor. High priority work can go to another. Parallelism is added by adding more actors. Using a worker pool is another similar approach.

A lock is still necessary because the actors share state.

The advantage is the lock is kept inside one application so the application is totally responsible for locking behaviour. Another subsystem couldn't spike your times.

The disadvantage is this architecture is more complex for applications to create. Message order issues also arise if parallel work is spread across multiple actors.

Event Queue Model

In the event queue model work is represented by events. Events are queued up to one thread or sometimes a few threads of different priorities. Applications don't have their own thread. All application work is queued up in event threads. Applications are usually asynchronous.

Obviously with multiple threads deadlock and locking issues still apply. One thread is safer but other issues arise. The single thread issues are priority and latency. The same sort of issues that are already addressed in thread scheduling system, which is why i've never really like the model because you have to write your own scheduler.

But, you still need cooperative multitasking, which makes it largely the same as the actor model.

Event model is attractive where most of the work is of approximately equal and small size. This model has been used very well in state machine oriented computation ([communicating finite state machines](#)) where state transitions require deterministic and small time. Breaking down your system to make it suitable for this model is big job. Paybacks are reduced complexity of implementation. Message or event priorities can be used in this model. This model is usually based on run to completion kind of scheduling.

If latency is an important issue the event model can't be used, at least everywhere as you'll never be able to guarantee latency. Attempts to guarantee latency basically require writing another level of scheduling.

Actor + Parallel State Machine Model

If synchronous logic is used then an actor can not process high priority messages when a low priority message is being processed.

One alternative is to use a state machine model in the processing of the request. The message is sent to the other actor and the reply is expected on a specific message queue. Now the actor begins the processing of the high priority message. Next time it picks up a message it can look for the reply message and resume processing of the low priority message. Having a separate message queue simply makes looking for the reply rather trivial.

In short, having multiple message queues for different priority messages combined with state machine driven message processing when required can potentially give, in many situations, low latency processing of high priority messages.

For example, msg1 comes at high priority, then msg2 comes in at medium priority, and then comes msg3 at the low priority. The assignment of these priorities to different requests is part of application protocol design. The heuristics used by an application are as follows. Process msg1 first. If more than 90 ms is spent processing msg1 and msg2 is starving, process one msg2 for 10 ms and then go back to msg1 processing.

If a msg3 is pending and it has not been processed for the last 1 minute, process msg3. Note that in this example each message is processed to completion. The heuristic kicks in to avoid starvation when large number of requests comes in.


The state machine is parallel in terms of the number of requests it can take. For each request processed you create a context for the request and the context is driven by the state machine. You might, for example, get a flood of msg1s. For each msg1 being processed, there will be a context block for storing an id (for context identification purpose), current state, and various associated data. As hundreds of msg1s are processed, hundreds of these contexts are created. One of the state these contexts go through is communication to another node. There is a timer

associated with each context in case reply times out. As ACK comes in or the timer fires, the associated context makes its state transition.

Many applications can be implemented in a straight forward fashion in this model to achieve high throughput and low latency for high priority work.

Starvation, deadlock, dependencies are application protocol layer issues. One can build tools to describe the dependencies with a high level descriptive syntax of some sort and have the underlying infrastructure implement them.

Dataflow Model

A technique used in [functional languages](#) . It allows users to dynamically create any number of sequential threads. The threads are dataflow threads in the sense that a thread executing an operation will suspend until all operands needed have a well-defined value.

Erlang Model

<http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf> .

[Erlang](#) has a process-based model of concurrency with asynchronous message passing. The processes are light-weight, i.e require little memory; creating and deleting processes and message passing require little computational effort.

No shared memory, all interaction between processes is by asynchronous message passing. The distribution is location transparent. The program does not have to consider whether the recipient of a message is a local process or located on a remote Erlang virtual machine.

SEDA Model

SEDA is an acronym for staged event-driven architecture, and decomposes a complex, event-driven application into a set of stages connected by queues. Each queue has a thread pool to execute work from the queue.

This design avoids the high overhead associated with thread-based concurrency models, and decouples event and thread scheduling from application logic. By performing admission control on each event queue, the service can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA employs dynamic control to automatically tune runtime parameters (such as the scheduling parameters of each stage), as well as to manage load, for example, by performing adaptive load shedding. Decomposing services into a set of stages also enables modularity and code reuse, as well as the development of debugging tools for complex event-driven applications.

Multiple Application Thread Pool Model (M - N)

In this model multiple applications are multiplexed over the the same thread pool. The pool may have only one thread. This could be considered a container for the parallel state machine or event model. See [App Backplane](#) for an exploration of this "best of all worlds" approach.

Where is the state machine?

Every application can be described by an explicit or implicit [state machine](#). In the actor model the state machine is often implicit on the stack and in the state of the actor. Clearly an actor can use a state machine, but it is often easier for developers to not use a formal state machine.

In the event and parallel state machine models state machines are usually made explicit because responses are asynchronous which means the response needs to be matched up with the original request. A state machine is a natural way to do this.

Applications built around state machines tend to be more robust.

Remote Queue Case Study

The Remote Queue is part of Data Grid library and supports peer-to-peer communication channels.

A Remote Queue supports these operations:

- register
- signal timer
- register / get next timer
- get next operation
- get next batch from a peer
- deregister
- available
- signal
- get status
- resolve other poller name space
- add data source

If a Remote Queue was an actor then if an application on a node needs to talk to 30 other nodes then there are 30 threads created. If want to use more peer-to-peer channels then the number of threads would increase proportionately.

Let's say there about 200 potentially parallel operations with 30 remote queues.

1-1 Threading

The advantage of the actor per queue model is that all queues are completely independent and the actor is the state machine for operations. Operations for one queue won't block another queue. Queues can have different priorities.

We'll have 30 simultaneous requests which make create [flow control?](#) problems, especially as the number of threads is increased. For example, we could have 30 simultaneous registration requests. That uses a chunk of memory, but imagine 200 simultaneous messages would use a lot of memory.

1-N Threading

Consider an architecture where all 30 queues multiplex over one actor. No shared locks. Smallest number of threads.

Part of the work is to dispatch data to applications where upon the applications will process the data. If this is done then the queue is blocked on applications, which means it could take an arbitrary period of time for the application to complete its work. This means all the other queues aren't be serviced.

We don't have the ability to gurantee latency or use priority. Other queues are being starved.

We could say to each application, "be really fast," but the won't work very well.

M-N Threading

We can have multiple queues assigned to multiple actors. This reduces the blocking factor because each pool of queues will their own thread.

Introduces shared locks. Has the same problems has the 1-N solution, though it may be less severe. We are not starving as many queues.

Parallel State Machine Model

We could have each register command be a state machine that sent a register request and wait for an async reply. We could make all 200 operations parallel, which is quite a lot. The remote queue would have to limit the amount of simultaneous work somehow.

Application processing of polled data is in the queue thread so this would need to be made async as well.

The issue with this approach is always how. The actor/synchronous style is easy to explain and do.

The [App Backplane](#) tries to make this style doable at the application level while handling latency, fairness, dead lock, etc.

References

- [What Is Wrong With Threads](#)
- [Handling Infinite Work Loads](#)
- [Scalability Problems/Scalability Solutions](#)
- [App Backplane](#)
- [seda](#)
- [Erlang](#)
- [Why Events Are A Bad Idea](#) ↗
- [Leader/Followers](#) ↗
- [Adaptive Communication Environment](#) ↗
- [Notes on Distributed Computing](#) ↗
- [Cycle Free Scheduling](#) ↗
- [Jet Propulsion Architecture](#)
- [The Pillars of Concurrency](#) ↗ - Building a consistent mental model for reasoning about concurrency