On The Duality of Operating System Structures
The Basic Idea

Two common ways to structure systems:
- message oriented (i.e., event-based)
- procedure oriented (i.e., multi-threaded)

They are duals of each other
- any program written in one form can be automatically transformed into the other form with no loss of functionality
- neither one is inherently better than the other
Message Oriented Systems

Small number of long-lived processes

Communication via message channels/queues
- each process extracts a request from its queue
- processes it before extracting the next request
- no shared data!
Message Oriented Systems

This is essentially event-based programming
- no shared data, no need to synchronize access to shared data
- reactive execution model using handler functions
- each request is executed to completion
Procedure Oriented Systems

Global shared data with accesses from multiple threads synchronized using monitors
- monitors = locks and condition variables

Thread per task model
- threads may be short-lived (created to handle a single task)
- they can access any data
- they communicate using shared data
The Duality Argument

What encapsulates the data?
How do control flows synchronize their access to data?
How does communication occur?
How does a control flow block and resume?
What is the real work and how can we separate it from the synchronization framework?
The Duality Mappings

Process == monitor
Message channel == monitor entry method
send message, wait for reply == entry method call
asynchronous send, work, wait == fork, work, join
send reply == return
main loop, wait for message == lock, entry method
wait for message == wait on condition variable
send message == signal condition
The Duality

One form can be converted into the other via a straightforward program transformation

The logic of the program itself is unchanged

The structure of the program is unchanged
The Duality

Any visual difference is purely syntactic

- computation is triggered by SendMessage in one model and Fork or Signal in the other

- computation is blocked by AwaitReply or WaitForMessages in one model and by Join or WaitCondition in the other

- data access is serialized by the WaitForMessages loop in one model and by a Monitor Mutex in the other
Conclusion

Disagreement between the two sides seems to be based more on emotion than fact
- there really isn’t a fundamental difference between these two models

Which model should you use?
- maybe a particular architecture has better hardware support for one model than the other
- the choice may depend on the underlying platform