ON THE DUALITY OF OPERATING SYSTEM STRUCTURES

Paper by Hugh C. Lauer and
Roger M. Needham
Presentation by Arthur Peters

Basic thesis

- There are 2 basic types of parallel systems:
  - Message-oriented systems: based on passing messages among multiple independent processes.
  - Procedure-oriented systems: based on communicating via locks and shared state.
- They are duals of one another in both capabilities and performance.

Paper Structure

- The paper is designed to be an “empirical paper”, that is a paper based on examples not rigorous models.
- The paper is divided into 3 sections:
  1. A detailed description of the Message-oriented system design
  2. A detailed description of the Procedure-oriented system design
  3. A comparison of the two and a description of the mapping between the two.

How are the models similar?

- Duality: A program or system written using one model can be translated into the other model by replacing one set of primitives with another.
  - Not based on simulation but on actual program transformation.
- Textual similarity: A program and it's dual can be made very similar textually.
- Performance: The performance of the dual systems are comparable.

Which model to use?

- The models are so similar that the choice should be based on something outside.
  - Underlying hardware substrate.

The middle ground

- Anecdotally it is shown that merging the 2 methods does not work well.
- It is stated that systems that do not roughly match one of the models are: “ill-structured and unstable” and are not usable or manageable. (This is also stated with no proof of any kind)

Message-oriented systems

- Specific communication channels are established between pairs of processes.
- Messages are passed along these channels.
- The number of processes and the connections between them stay relatively static.
- Each process has it's own static context.
- It is difficult to create and destroy processes.

Message-oriented: Good design

- Synchronization among processes and queuing for congested resources is implemented in the message queues.
- Shared data structures are passed (by reference) in messages. Ownership goes with the message.
- Peripheral devices are treated as processes.
- Priorities tend to be statically assigned to processes.
- Processes handle messages roughly serially.

Message-oriented: Standard process definition

```
begin m: messageBody;
      i: messageId;
      p: portid;
      s: set of portid;
      Initialize; ... -local data and state information for this process
      do forever;
            [m, i, p]«- WaitForMessage[s];
            case p of
                  port1 =>...; -algorithm for port1
                  port2 =>...
                        if resourceExhausted then s = s - port2;
                        SendReply[i, reply];
                        ...; -algorithm for port2
                  portk =>  ...
                        s = s + port2 ...; -algorithm for portk
            endcase;
      endloop;
end.
```

Message-oriented: Conclusions

- Overall this system provides some notable advantages:
  - Each process can run in it's own separate address space.
  - Because messages are processed in sequence, locks or other forms of protection are not needed.
- Notably the paper does not mention that the processes can still dead lock against each. Especially because of the AwaitReply construct.

Procedure-oriented systems

- Global data can be both protected and efficiently accessed by providing procedural interfaces which do all of the synchronization.
- A process typically has only one goal or task, but it wanders all over the system.
- As a result, the system resources tend to be encoded in global data structures.
- It is easy to create and destroy processes.
- Procedures can be called in a non-blocking fashion with FORK & JOIN.

Procedure-oriented: Good design

- Synchronization of processes and queuing for resources occurs in the form of locks.
- Data is shared directly among processes, and protected by locks.
- Control of and 'interrupts' from peripheral devices take the form of manipulating shared data (including locks).
- Process priorities are associated with the locks or data structures.

Procedure-oriented: Standard process implementation

```
ResourceManager: MONITOR =
    C: CONDITION;
    resourceExhausted: BOOLEAN;
    ... "global data and state information for this process
    prod : ENTRY PROCEDURE[ . . . ] =
        ...; "algorithm for prod
    proc2: ENTRY PROCEDURE[ . . . ] RETURNS[ . . . ] = BEGIN
        IF resourceExhausted THEN WAIT c;
        RETURN[results];
        ...;
    END; "algorithm for proc2
    procL: ENTRY PROCEDURE[ . . . ] = BEGIN
        resourceExhausted«- FALSE;
        SIGNAL C;
        ...;
    END; "algorithm for procL endloop;
    initialize;
END.
```

Procedure-oriented: Conclusions

- No process can be associated with a separate address space because calls need to cross module boundaries.
- This system can also dead lock because of the standard non-ordered locks issue. (not mentioned in the paper.)

Mapping

- Processes, CreateProcess
- Message Channels
- SendMessage; AwaitReply
- SendMessage; …; AwaitReply
- Monitors, NEW External
- ENTRY Procedures
- Normal Procedure call
- FORK; …; JOIN

Mapping

- SendReply
- Main loop and WaitForMessage
- Waiting for messages
- RETURN
- Lock, ENTRY
- Condition variables, WAIT, SIGNAL

Mapping

- Interestingly, processes map to monitors.
- Programs can be converted from one form to the other by a transformation on the program.
- The main logic of the program is totally unchanged.
- The mapped version of a program could be made to look almost exactly like the original.

Mapping synchronization points

Triggering computation or action (unblocks another)

- SendMessage
- FORK or SignalCondition

Waiting for a reply or state change (blocking)

- AwaitReply or WaitForMessages
- JOIN or WaitCondition

Serializing access to internal data (blocking)

- WaitForMessages loop
- Monitor wrapping all procedures

Unmappable things

- The mapping requires that the program only use the canonical primitives and that they are written in the way outlined here.
- If the primitives are used in ways that are not listed here is that a really a good idea?

Performance

- The communication overhead is the same.
- For example:
  - Sending messages has the same performance as a FORKing a procedure.
  - Leaving a monitor is the same as waiting for a message
  - Context-switches and memory are the same. Because they are actually identical.
- If the message and lock queues use the same discipline, the timing will be the same.

Evidence

- It is rather hard to convert a system from one of the choices to the other because many things are bound into the design.
  - Global data use
  - Communication style
- So it's very hard to convert a real system.
- However there is one example: Cambridge CAP Computer

Conclusions

- A lot of people disagree with this.
- The argument between the sides has often been based more on emotion than facts.
- Both models are equal when it comes to performance, elegance and soundness.
- By treating them as equal you eliminate false choices from the system design process.
- It might be possible to develop a uniform model that unifies the models.