RESOURCE MANAGEMENT

William Bricken
April 1988

An operating system manages the resources of the system (processes and memories). There are distinct advantages in managing the resources and the message traffic in a Class-Instance hierarchy, also.

The question for serial processors is: "Which instance should have the process resources of the system at a given time?"

The question for parallel processors is: "How will the results of the many processes be integrated into a single result?"

Of course, these are the same question. A serial (or extremely process limited) system assigns the available processor to the specific object that contributes most to the integration of the final result. It's just that the serial model has no difficulty with global integration, that difficulty has been transferred (formally) to a question of *sequencing*. And sequencing (as are most difficulties) is passed off to the person doing the implementation, who has to worry about proper calling and branching structure in the code.

Some options for controlling resources in an object-oriented system:

Zero-ith

I'd expect the language to provide process-management primitives.

Serial-god

All messages get sent to a central, global MESSAGE HANDLER, which determines what goes where, which are syntactically in error (ie: policing the traffic for permitted mail by type checking), and in what order messages get distributed. Handles creation and destruction messages (memory management), and determines assignment of processors by providing each message with a processing capability. Might choose to interrupt an object if it doesn't relinquish process control.

The issue here is how the Serial-god manager chooses to switch it attention to different objects. If an object relinquishes processing by sending a message, the decision is easy, cause control is returned to the global context. The manager can use its own priorities to select the next message to process. If the object *branches*, by sending out messages without relinquishing processing power, then the decision is difficult. How will the Serial-god know the importance and impact of the branching messages on the ongoing process of the active object?

Succinctly: the Serial-god model constrains Methods by separating control logic (if, do, loop) from message sending. Method code cannot include both control and send. Technically, objects can be Terms and Relations, but not longer Sentences.

Serial-mailbox

All messages go to a central mailbox, which then distributes them to the appropriate objects. The mailbox can have a distribution priority, but priorities are not motivated by reaching the end of the computation. Rather, the mailbox can enforce the local resource constraints of the operating environment. For example, if there is a CREATE message to be delivered, and no more memory, the manager will hold that message until it can assign memory.

Within this model, the Mailbox is responsible for memory management only, and will defer process management to the local contexts. This means that the flow of control is dictated by the flow of messages. Again, branching messages create a problem, they must form into a stack and await the completion of their predecessors. For example, (if A then B else C): C will have to wait for B to complete.

Type and syntax checking is done locally, as an object selects a message to process. The object can have a local priority for selection of incoming messages, and it can be assured that messages using memory have pre-allocated space.

Memory-manager

Here, the messages are not routed through a global mailbox. Rather, each object has local knowledge of its connectivity and has the address of each other object it can communicate with. Messages are sent directly.

Variations on this theme are common: A Class object, for instance, can contain all its instances. Messages from instances are always set to the Class for distribution. Another possibility is for objects to be able to infer the path of indirect messages. The regime is: "If this is not going to you, then send it to who it is going to. If you don't know, send it to someone who you think might know."

The main idea, though, is that only CREATE and DESTROY messages, ones that directly effect memory management are sent to the global Memory-manager. Every local node knows the manager, and restricts memory transactions to that channel. The Memory-manager is more like a specialized object than a global dictator.

The problem with this is that the local objects need to be smart about global processing needs. When they send a CREATE message, they must stop and wait for space. If memory is in short supply, or if the Memory-manager is very busy, objects get stuck.

Intelligence about global processing can be built into the structural connections between objects (for some theories). Rather than envisioning full connectivity between objects, the *relational structure* can be stored in the connectivity lists. This is a nice way to solve some database problems.

Parallel-process-manager

Modularity, composability, portability, and the other advertised advantages of object-oriented programming are all due to a single characteristic of the model:

asynchronous communication between independent, autonomous processes

This mouthful describes objects with processing power of their own, making their own control decisions, and sending messages that are both independent of timing and purposive in content. If you have a theory that meets these characteristics, it will also have all the characteristics desired in beautiful implementation code.

There are several variations of Parallel-process-managers. We could imagine each object with sufficient local memory that *no* global manager is necessary at all. We could imagine a Shared-memory-manager that polices memory access and modification. We could imagine a Process-manager that distributes the process pool across many objects (perhaps by demand), or a Process-middle-manager that acquires free processing power from idle objects and brokers it to busy objects.

The main idea, though, is that we should adopt a mathematical architecture that accommodates all of these management options. The organization of our software must permit many structural manifestations *without requiring organizational change*.

Notes

The deep issue: *where do we focus our attention?* There are two distinct but complementary perspectives for object-oriented programming. We can form a conceptual model of the Hierarchy, the global configuration, and we can form an experiential model of the Specific Object, the local configuration. Traditional programming styles place the implementer in the Global-god position. In procedural languages, we orchestrate (code) by giving instructions to the processor. Our conceptual model of processing is the

assembly of little actions; we take responsibility for data and process.

The complementary (object-oriented) perspective is more humble. We take responsibility for a small locale, a specific Object. The Object has the capability for both structure (state) and process (methods), but the interest is purely local. The Object must ask other objects, which are, by definition and by location, in unknowable states. The object can ask "What's your temperature?" And when it gets a reply, the reply must be *time-stamped*. "My temperature at moment=12843 is 40." An Object can rely on simultaneous knowledge only when the information provider agrees to freeze shared information.

We trust the structure of the model to handle global coordination.

This shift in thinking is similar to that made in declarative languages such as Prolog. In logic programming, we abandon all responsibility for processing, including both branches and ordering. All we must do is *describe* the domain. We run logic programs by submitting questions. We trust the structure of logic (implemented in the inference engine) to handle global coordination.

The problem is that logic is not smart enough (yet, but wait a few years...) to be as smart as the implementer about efficiency questions. So the order of our descriptions (code) does effect how long it takes for our answer to return. It's the same for database search questions. For instance, finding the President with the most children is much easier if we search the list of Presidents to find the kids, rather than searching the list of Kids to find if their parent was a president. And logic will continue to the end of certain branches of enquiry, when an implementer will know when to stop early.

The shift in thinking is similar to that made in functional languages such as Pure LISP. In functional programming, we abandon all responsibility for State, by using a language that keeps everything important to the result in the current process. Functional programs are purely purposive; we run programs by submitting values. We trust the structure of function composition to coordinate transformation of values into results.

Summary

My opinion is that different applications, different mathematical theories that reflect conceptual models, require different implementation languages. Generally, the languages at the machine level are cross-translatable, but thinking in a disfunctional metaphor (language) is disasterous.

We need to spend our time:

1. Building a conceptual model of CAD functionality.

2. Writing tools that permit conceptual models to be expressed by mathematical organizations, and that permit implementation independent syntax.

3. Optimizing the implementation of the fundamental computational techniques (pattern-matching, substitution, display).