Because the original of the following paper by Lauer and Needham is not widely available, we are reprinting it here. If the paper is referenced in published work, the citation should read: "Lauer, H.C., Needham, R.M., "On the Duality of Operating Systems Structures," in Proc. Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

On the Duality of Operating System Structures

Hugh C. Lauer
Xerox Corporation
Palo Alto, California

Roger M. Needham*
Cambridge University
Cambridge, England

Abstract

Many operating system designs can be placed into one of two very rough categories, depending upon how they implement and use the notions of process and synchronization. One category, the "Message-oriented System," is characterized by a relatively small, static number of processes with an explicit message system for communicating among them. The other category, the "Procedure-oriented System," is characterized by a large, rapidly changing number of small processes and a process synchronization mechanism based on shared data.

In this paper, it is demonstrated that these two categories are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other. The principal conclusion is that neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built, not the application which the system will ultimately support.

This is an empirical paper, in the sense of empirical studies in the natural sciences. We have observed a number of samples from a class of objects and identified a classification of some of their properties. We have then generalized our classification and constructed abstract models to describe these properties. With the aid of these models, we were able to make some observations about the nature of the objects themselves, observations which are supported by other experimental evidence. Finally, we have drawn some conclusions about the class of objects which better aid our understanding of that class and the decisions which affect the design of members of that class.

The universe in this investigation is the class of operating systems, and the properties in which we are interested are the ways in which the concepts of process, synchronization, and interprocess communication occur within these systems and among their clients. There appear to be two general categories in this respect, which we designate the *Message-oriented Systems* and the *Procedure-oriented Systems.* Most systems which we have observed tend to be biased fairly strongly in favour of one or the other, rather than being neutral or indeterminate. Moreover,

* This work was done while the author was on sabbatical leave at the Xerox Palo Alto Research Center during the summer of 1977.

within each of the categories, the systems tend to be more like each other than like systems of the other category. Finally, in several design efforts in which either of us have participated or observed first-hand, attempts to combine fundamental characteristics from the two categories have met with failure or have been abandonned.

To characterise our classifications, we have constructed a *canonical model* for each category. The message-oriented model is characterised by a small, relatively static number of big processes, an explicit set of message channels between them, a relatively limited amount of direct sharing of data in memory, and an identification of address space or context with processes. The procedure-oriented model is characterized by a large number of very small processes, rapid creation and deletion of processes, communication by means of direct sharing and interlocking of data in memory, and identification of the context of execution with the function being executed rather than with the process. These two models define two differents kinds of primitive operations for managing processes and synchronization in an operating system. From them, we will derive three observations:

1. The two models are duals of each other. That is, a program or subsystem constructed strictly according to the primitives defined by one model can be mapped directly into a *dual program or subsystem* which fits the other model.
2. The dual programs or subsystems are logically identical to each other. They can also be made textually very similar, differing only in non-essential details.
3. The performance of a program or subsystem from one model, as reflected by its queue lengths, waiting times, service rates, etc. is identical to that of its dual system, given identical scheduling strategies. Furthermore, the primitive operations provided by the operating system of one model can be made as efficient as their duals of the other model.

The principal conclusion we will draw from these observations is that the considerations for choosing which model to adopt in a given system are not found in the applications which that system is meant to support. Instead, they lie in the substrate upon which the system is built and are a function of which set of primitive operations and mechanisms are easier to build or better suited to the constraints imposed by the machine architecture and hardware.

In the remainder of the paper, we develop the canonical models in greater detail. We then present the three observations, our reasons for believing them, and some empirical support for them. Finally, we discuss the consequences and conclusions which we derived from this point of view.

Two Models

It is not helpful in this paper to develop an elaborate formalism defining the two canonical models for our categories of operating systems. Instead, we will describe these in informal English, outlining the characteristics of each in familiar terms. Similarly, our observations will be based on

informal arguments about the models, not on formal, rigourous proofs.

An important caveat to bear in mind is that these models bear roughly the same relationship to reality as, for example, the postulate of frictionless surfaces does to reality in physics. That is, no real system precisely agrees with either model in all respects. Furthermore, most operating systems typically have some subsystems which behave like one and other subsystems which behave like the other. Thus the observations which we will make apply only to our models, and our conclusions will describe a real system only to the same degree that that system corresponds to one or the other of the models.

However, we believe and have observed that the models are reasonable in the sense that most modern operating systems can be usefully classified using them. Some systems are implemented in a style which is very close in spirit to one model or the other. Other systems are able to be partitioned into subsystems, each of which corresponds to one of the models, and which are coupled by explicit interface mechanisms. Most of the remaining systems are so ill-structured and unstable (for example, the ratio of bits, operations, and/or interfaces to information content is much too high) that they are unreliable, unmanageable, uneconomic, and unusable.

Message-oriented System

At the level of mechanism, this kind of system is characterized by facilities for passing messages (or events, or whatever they might be called in a particular system) easily and efficiently among processes. There also convenient facilities for queuing messages at destination processes until they can be acted upon. Processes are provided with primitive operations to send messages, wait for any message, wait for a particular class of message, and examine the state of the message queue. Pre-emption of the processor occurs when a message arrives at a 'higher priority' process which is waiting for a message of that kind.

Some of the hallmarks of successful systems designed according to this model are the following.

> Specific communication paths (i.e., message channels, ports, sockets, or other means of identifying classes of messages) are established for specific forms of communication between particular pairs of processes. This binding typically persists for relatively long periods, and is often done when the system is initialized.
>
> The number of processes and the connections between them remain relatively static. Deletion of processes tends to be very difficult because of the possibility of an arbitrary number of queued messages awaiting response. Creating processes and changing connections can be correspondingly difficult.
>
> Each process tends to operate in a relatively static context. Virtual memories or address spaces are usually placed in one-to-one correspondence with processes. Processes rarely cross protection boundaries (except to briefly enter the executive or kernel), and they rarely share data in memory.

As a result, processes tend to be associated with system resources, and the needs of applications which the system exists to serve are encoded into data to be passed around in messages.

This style of system architecture is most common in the world of real-time systems and process control, where frequently the applications themselves are encoded in message blocks denoting transactions. However, there are also a number of general purpose operating systems implemented in this way, including, for example, IBM's OS/360[1]. The most elegant example of a message-oriented system is the GEC 4080[2].

In this style of system, a number of characteristics tend to emerge as natural consequences of good design practice:

Synchronization among processes and queuing for congested resources is implemented in the message queues attached to the processes associated with those resources.

Data structures which must be manipulated by more than one process are passed (by reference) in messages. No process touches the data unless it is currently processing a message referring to it, and a process does not continue to manipulate the data after it has passed it on in a message to another process.

Peripheral devices are treated as processes (or virtual processes). Control of a device often resembles sending a message to that device, and an 'interrupt' from the device is manifest as a message to some other process.

Priorities tend to be statically assigned to processes at the time the system is designed, and they correspond to the timing needs of the resources being managed.

Processes operate upon one or a very small number of messages at a time and normally complete those operations before looking at the message queues again.

Because processes operate in static contexts, neither procedural interfaces no global naming schemes are very useful.

Our canonical model is, therefore, and idealized operating system kernel and/or programming environment which provides the following facilities:

Messages and message identifiers. A *message* is a data structure meant for sending information from one process to another; it typically contains a small, fixed area for data which is passed by value and space for a pointer to larger data structures which must be passed by reference. A *message identifier* is a handle by which a particular message can be identified.

Message channels and message ports. A *message channel* is an abstract structure which identifies the destination of a message. A *message port* is queue capable of holding messages of a certain class or type which might be received by a particular process. Each message channel must be bound to a particular message port before is can be used. A message port, however, may have more than one message channel bound to it.

Four message transmission operations:

SendMessage[messageChannel, messageBody] returns [messageId] -- This operation simply queues a new message on the port bound to the the **messageChannel** named as parameter. The **messageId** returned is used as parameter to the following operation.

AwaitReply[messageId] returns [messageBody] -- The operation causes the process to wait for a reply to a specific message previously sent via **SendMessage.**

WaitForMessage[set of messagePort] returns [messageBody, messageId, messagePort] -- This operation allows a process to wait for a new (unsolicited) message on any one of the message ports named in the parameter. The message which is first on the queue is returned, along with a message identifier for future reference and an indication of the port from which that message came.

SendReply[messageId, messageBody] -- This operation sends a reply to the particular message identified by the message identifier.

Process declarations. A *process* (or more precisely, a *process template*) consists of local data and algorithms, defines certain message ports, and refers to (i.e., sends messages to) certain message channels representing other processes.

The operation **CreateProcess.** This operation creates (an instance of) a process which has been previously declared, and binds the message channels it references to message ports of previously existing processes. Note that because of the binding, this operation is rather cumbersome and should not be used extravigantly. (No **DeleteProcess** operation is provided in our model, because it would be messy and not important.)

Finally, our canonical model for the message-oriented system suggests a standard way, characterised by the program outline below, for implementing a simple resource manager using these primitive system operations. It consists of a single process containing local data to represent the state information necessary for managing that resource and a loop which waits for a request from any of a set of ports, then services that request.

```
begin m: messageBody;
        i: messageId;
        p: portId;
        s: set of portId;
        . . . --local data and state information for this process
        initialize;
        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.
```

In this process, the kind of service requested is a function of which port the requesting message arrives on. It may or may not involve making requests of still other processes and/or sending a

reply back to the requestor. It may also result in some circumstance, such as the exhaustion of a resource, which prevents further requests from being considered. These remain queued on their port until later, when the process is willing to listen on that port again.

Note that if a whole system is built according to this style, then the sole means of interaction among the components of that system is by means of the message facility. Each process can operate in its own address space without interference from the others. Because of the serial way in which requests are handled, there is never any need to protect the state information of a process from multiple, simultaneous access and updating.

Procedure-oriented System

At the level of mechanism, this kind of system is characterized by a protection and addressing mechanism oriented toward procedures and efficient procedure call facilities which can take a process very rapidly from one context to another. Cooperation among processes is achieved by some form of locks, semaphores, monitors, or other synchronizing data structures (we will use the term *lock* as a generic identification of these). In this kind of system, a process attempts to claim a lock, and may be forced to wait on a queue until some other process releases it. Pre-emption of the processor occurs when a release operation is performed on a lock which a 'higher priority' process is attempting to claim.

Some of the hallmarks of successful system design in this environment are the following:

> Global data can be both protected and efficiently accessed by providing procedural interfaces which do all of the synchronization and manipulation in controlled ways.
>
> Process creation is very easy since no communication channels have to be set up with existing processes; deletion of a process is correspondingly easy so long as it is not holding any locks.
>
> A process typically has only one goal or task, but it wanders all over the system (by means of calling procedures to enter different contexts) in order to get that thing done.
>
> *As a result, the system resources tend to be encoded in common or global data structures and the applications are associated with processes whose needs are encoded in calls to system-provided procedures which access this data.*

This style is characteristics of a wide variety of designs, including HYDRA[3], the Plessey System 250[4], and others.

Some of the characteristics of systems which result from this viewpoint are the following:

> Synchronization of processes and queuing for congested resources occurs in the form of queues of processes waiting for locks associated with the corresponding data structures.
>
> Data is shared directly among processes, and processes tend to lock only small parts of the data structures for relatively short periods of time.

Control of and 'interrupts' from peripheral devices take the form of manipulating locks and/or shared data in memory.

Processes inherit their priorities dynamically from the contexts in which they execute; priorities are associated with the locks or data structures and correspond to the timing requirements of the resources they represent.

Global naming schemes are an important feature in optimizing the context switching, and the contexts represent an import form of protection.

Our canonical model for a procedure-oriented system is an operating system kernel and/or a programming environment which provides the following facilities, which we describe as hypothetical extensions to Mesa, a Pascal-like language developed at Xerox[5]:

Procedures. A *procedure* is a piece of Mesa text containing algorithms, local data, parameters, and results. It always operates in the scope of a Mesa module and may access any global data declared in that module (as well as in any containing procedures).

Procedure call facilities, synchronous and asynchronous. The *synchronous* procedure call mechanism is just the ordinary Mesa procedure call statement, which may return results. This is very much like procedure or function calls in Algol, Pascal, etc. The *asynchronous* procedure call mechanism is represented by the **FORK** and **JOIN** statements, which are defined as follows:

processId ← FORK procedureName[parameterList] -- This statement starts the procedure executing as a new process with its own parameters. The procedure operates in the context of its declaration, just as if it had been called synchronously, but the process has its own call stack and state. The calling process continues executing from the statement following the **FORK**. The process identifier returned from **FORK** is used in the next statement.

[resultList] ← JOIN processId -- This statement causes the process executing it to synchronize itself with the termination of the process named by the process identifier. The results are retrieved from that process and returned to the calling process as if they had been returned from an ordinary procedure call. The **JOIN**ed process is then destroyed and execution continues in the **JOIN**ing process from the statement following the **JOIN**.

Modules and monitors. A *module* is the primitive Mesa unit of compilation and consists of a collection of procedures and data. The scope rules of the language determine which of these procedures and data are accessible or callable from outside the module. A *monitor* is a special kind of Mesa module which has associated with it a lock to prevent more than one process from executing inside of it at any one time. It is based on and very similar to the monitor mechanism described by Hoare[6].

Module instantiation. Modules (including monitor modules) may be *instantiated* in Mesa by means of the **NEW** and **START** statements. These cause a new context to be created for holding the module data, provide the binding from external procedure references within the module to procedures declared in other modules, and activate the initialization code of the module.

Condition variables. *Condition variables* are part of Hoare's monitor mechanism an provide more flexible synchronization among events than mutual exclusion facility of the monitor lock or the process termination facility of the **JOIN** statement. In our model, a condition variable, must be contained within a monitor, has associated with it a queue of processes, and has two operations defined on it:

WAIT **conditionVariable** -- This causes the process executing it to release the monitor lock, suspend execution, and join the queue associated with that condition variable.

SIGNAL **conditionVariable** -- This causes a process which has previously WAITed on the condition variable to resume execution from its next statement when it is able to reclaim the monitor lock.

Note that because the FORK and JOIN operations apply to procedures which are already declared and bound to the right context, these operations take the same order of magnitude of time to execute as do simple procedure calls and returns. Thus processes are very lightweight, and can be created and destroyed very frequently. Module and monitor instantiation, on the other hand, is more cumbersome and is usually done statically before the system is started. Note that this canonical model has no module deletion facility.

As we did for the previous model, we can define a standard style of simple resource manager for the procedure-oriented system. This is characterised by the program outline below. It consists of a monitor containing global data representing the state information for the resource, plus a number of procedure declarations representing the different services offered.

```
ResourceManager: MONITOR =

    c: CONDITION;
    resourceExhausted: BOOLEAN;
    . . . --global data and state information for this process

    proc1: ENTRY PROCEDURE[ . . . ] =
        . . . ; --algorithm for proc1

    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.
```

The attribute ENTRY is used to distinguish procedures which are called from outside the monitor, thus seizing the monitor lock, from those which are declared purely internal to the monitor. Any

of the procedures in this module may, of course, call procedures declared in other modules for other system services before returning. Within the monitor, condition variables are used to control waiting for circumstances such as the availability of resources. These are used in this standard resource manager to control the access of a process the procedure representing a particular kind of service.

If a whole system is built in this style, then the sole means of interaction among its components is procedural. Processes move from one context to another by means of the procedure call facility across module boundaries, and they use asynchronous calls to stimulate concurrent activity. They depend upon monitor locks and condition variables to keep out of the way of each other. Thus no process can be associated with a single address space unless that space be the whole system.

Characteristics of the Models

The importance of these rather idealized models is that we can show that the two styles of system design are duals of each other. We will show how a program for one kind of system can be mapped into a program appropriate for the other. We will also show that as a result of this mapping, the logic of the programs in the dual systems is invariant. Finally, we will argue that the performance of the system can be preserved across the mapping. It should be noted that our transformation is *not* the naive technique of simulating one set of primitives in terms of the other. Instead, it is a direct transformation on the programs themselves, exchanging the primitive operations and data structures of one style for those of the other.

The Duality Mapping

The mapping is derived from the following correspondence between the basic system facilities and canonical styles for resource managers:

Message-oriented system	*Procedure-oriented system*
Processes, **CreateProcess**	Monitors, **NEW/START**
Message Channels	External Procedure identifiers
Message Ports	**ENTRY** procedure identifiers
SendMessage; AwaitReply (immediate)	simple procedure call
SendMessage; . . . AwaitReply (delayed)	**FORK; . . . JOIN**
SendReply	**RETURN** (from procedure)
main loop of standard resource manager, **WaitForMessage** statement, **case** statement	monitor lock, **ENTRY** attribute
arms of the **case** statement	**ENTRY** procedure declarations
selective waiting for messages	condition variables, **WAIT**, **SIGNAL**

That is, the facilities on the left serve the same purpose in the message-oriented system as do the ones opposite them in the procedure-oriented system. For example, a **SendMessage** operation followed by an **AwaitReply** operation some time later is used by message system clients where a procedure system client would use **FORK** and **JOIN**.

The most interesting correspondence is between the processes of the one system and the monitors of the other. In particular, in the canonical style of resource manager, the arms of the **case** statement in the message-oriented model correspond to the **ENTRY** procedure declarations in the other. The code representing the main loop of the process, with its **WaitForMessage**, and **case** statement performs exactly the same function as the mutual exclusion lock on the monitor, namely that of serializing the requests for service and admitting only one at a time. The **case** statement itself sorts out which service is requested in the same way as the different procedure names do in the monitor.

Similarity of Programs

This forms the basis of the duality mapping. If a client system or subsystem is written in the strict style of one of our standard resource managers, then it can be transformed directly into the other kind of system by replacing each construct with its corresponding one. For example, each monitor is replaced with a process declaration containing a main loop in the style we suggested. All synchronous and asynchronous procedure calls are replaced by **SendMessage** and **AwaitReply** operations, and returns from procedures are replaced by **SendReply** operations. The use of condition variables for managing the synchronization of events is replaced by carefully selected waiting for messages. This transformation can, of course, be applied in either direction.

Note that by applying the transformation, we do not affect the logic of the client programs at all. In fact, none of their interesting parts are touched or even rearranged (except for the initialization code of each resource manager, which by tradition appears after the procedure declarations in the monitor but before the main loop in the process). No algorithms are changed; no data structures are replaced; no interface strategies are affected. Only the actual text representing the interactions between client system components is modified, and this is only to reflect the 'syntactic' details of the primitive facilities of the other kind of system. The semantic content is invariant.

This simplistic mapping between the two types of systems does not work if the systems being transformed do not adhere to the strict style we postulated. If a real system implements process or synchronization facilities which are very different from those of our canonical model, then the transformation must be extended and/or may not make sense. Similarly, if resource managers or users of resources are designed in a grossly different way, even though they use the same primitive operations, the transformation may produce a very contrived or awkward program structure. This raises an interesting question, to which we have no definite answer:

> If a primitive operation, process facility, or programming style is proposed for a system which fits into one of our two broad categories, and if no reasonable counterpart for it can be found in the other category, is it a good thing? That is, would a style or mechanism which has no dual be considered a truly well-structured construct which is elegant in form and rich in semantic content? Or would it be considered an overimaginative, ungainly feature which is awkward to program and hard to understand?

For example, the **WAIT** statement in the procedure-oriented model provides a considerably richer synchronization facility than does selective waiting for messages in the message-oriented model. In particular, a process may **WAIT** anywhere within the **ENTRY** procedure or any other monitor procedure called by it, not just at the beginning as we have suggested in the canonical style. However, this is not without its disadvantages. The procedure which **WAIT**s must ensure that the monitor invariant is true, even though it might be deep inside an inner block of an inner procedure and may have captured all sorts of monitor information and temporary results in its local variables, results which could easily be invalidated by another process entering the monitor. In this sense, the **WAIT** statement is almost as ill-structured as the notorious 'go to' statement. Perhaps, it should be confined to, say, the entry and exit points of an **ENTRY** procedure for more clarity.

One final observation with respect to the invariance of programs under the duality mapping: It is possible to imagine embedding the primitive synchronization and process facilities of the message-oriented system in a strongly-typed language such as Mesa. Furthermore, if we accept that the main loop of the canonical resource manager, the **WaitForMessage** operation, and the **case** statement are fundamental parts of the programming style, then they can be absorbed into the linguistic unit representing a process (so that a process declaration consists of some global data, and a set of actions to associate with each message port). Finally, we can make the sending of

messages look syntactically like calling or FORKing procedures, with the full typechecking facilities of the language applied to message bodies just as we apply it now to parameter and result lists. Then if we consider a system built according to the canonical style in this environment, we see that its dual is *textually identical except for the spelling of certain keywords of the language.*

Preservation of Performance

Our canonical models and standard styles for implementing resource managers suggest that there is another property which is invariant under the duality mapping, namely the performance of the client system. If we take due care in the implementation of the primitive operations of the two operating system kernels (and if we assume similar processor characteristics and peripheral devices), then a system of programs built in terms of one will have the same execution characteristics as its dual system built in terms of the other. To understand this, observe that there are three components of the dynamic behaviour of a system of programs:

> the execution times of the programs themselves,
>
> the computational overhead of the of the primitive system operations they call, and
>
> the queuing and waiting times which reflect the congestion and sharing of resources, dependence upon external events, and scheduling decisions.

The duality transformation leaves the main bodies of the programs comprising the system untouched. Thus the algorithms will all compute at the same speed, and the same amount of information will be stored in each data structure. The same amount of client code will be executed in each of the dual systems. The same number of additions, multiplications, comparisons, and string operations will be performed. Therefore if basic processor characteristics are unchanged, then these will take precisely the same amount of computing power, and this component of the system performance will remain unchanged.

The other component affecting the speed of execution of a single program is the time it takes to execute each of the primitive system operations it calls. We assert without proof that the facilities of each of our two canonical models can be made to execute as efficiently as the corresponding facilities of the other model. I.e.,

> Sending a message, with its inherent need to allocate a message block and manipulate a queue and its possibility of forcing a context (process) switch, is a computation of the same complexity as that of calling or FORKing to an ENTRY procedure, which involves the same need to allocate, queue, and force a context switch.
>
> Leaving a monitor, with the possibility of having to unqueue a waiting process and re-enter it, is an operation of the same complexity as that of waiting for new messages.
>
> Process switching can be made equally fast in either system, and for similar machine architectures this means saving the same amount of state information. The same is true for the scheduling and dispatching of processes at the 'microscopic' level.

> Virtual memory and paging or swapping can even be used with equal effectiveness in either model.

As evidence for this belief, we cite the GEC 4080[2], in which message queuing, process switching, and dispatching are implemented as fast operations in microcode, and an implementation of the Mesa system in which the dual operations are also implemented in microcode with similar speed. In general, we have observed that a message-oriented operating system kernel implemented by a dedicated team on a friendly machine architecture can be made very efficient relative to the basic cycle time of that machine. But we have also observed that the same is true for a procedure-oriented system if the machine architecture is appropriate for that. We can find no *inherent* differences in the two approaches.

Note that it is also possible to make the basic operations of the two models behave identically with respect to the scheduling and dispatching of client processes. That is, if the message system implements a particular discipline for queuing and unqueuing messages, then the procedure-oriented system can implement exactly the same discipline for its queuing and unqueuing of processes. Similarly, if one system forces a context switch in a particular circumstance, either as a result of a kernel operation or a pre-emption due to an external event, then the other model can do exactly the same in response to the dual circumstance. Thus, not only will operations happen just as quickly in one model as in the other, but corresponding events will happen in the same order.

This means that the third component affecting the performance of a suite of programs -- namely the way in which the executions of those programs interact with others -- is also invariant under the duality transformation, assuming that the two previous components are. Each message between processes (or between a process and a device) in the message-oriented system corresponds in the other system to a call to or return from a synchronous or asynchronous **ENTRY** procedure. If the message has to be queued because the destination process is not ready to receive it, then the procedure call will also be queued at its monitor, **WAIT** statement, or **JOIN** statement *for the same length of time.* The same external events will cause the congestion of a resource manager (either process or monitor) to be relieved at the same time. The peripheral devices will exhibit the same behaviour with respect to such issues as latency, response time, transfer times, etc., and thus the processes waiting for them to complete will wait just as long. Furthermore, the scheduling and dispatching can be arranged so that the same number of context switches, allocations of message blocks or local frames, etc., take place whether it is a message- or procedure-oriented system.

From these arguments, we claim that the total lifetime of a *computation* is the same for the two models, as is the juxtaposition of that lifetime with respect the lifetimes of other computations. In the procedure-oriented model, the computation corresponds to a process with its call stack. During its life, it wanders through the system, occasionally executing code, occasionally waiting in queues,

always crossing context boundaries by means of procedure calls and returns. But in the message-oriented model, the computation is represented only partly by the execution of code and partly by the transmission of messages. It also wanders through the system, occasionally executing the code of some process but always taking the form of a message when it crosses context boundaries or waits in queues.

Empirical Support

For accidental reasons, it is not very easy to change the structure of most operating systems in a way which would reflect the duality we suggest. The underlying addressing structures, use of global data, and styles of communication are usually so bound to the design and implementation that performing the transformation to a dual version would be a major exercise, not justified by the second order gains. Accordingly, there is not much evidence of example which can be quoted in support of our thesis.

However, one case can be cited, namely the Cambridge CAP Computer [7]. This system has a structure which leads to the complete addressing encapsulation of each system module, and to an operating system in which each module is implemented as a complete program (normally Algol68C). It was a basic design principle that any system data structure be managed by a single module (or protected procedure, in CAP terminology). An instance of such a procedure might be found in many or all of the processes in the system. Though this design approach was adopted for quite independent reasons, it turned out to facilitate just the kind of restructuring which we are discussing in this paper.

For example, the original design had a process which was devoted to the management of *system internal names*, which constituted a sort of central file directory referenced by all ordinary directories. This process was activated by messages to recover details of particular filed objects and to increment or decrement their reference counts. It was noticed that the message system was rather expensive, and accordingly the management program was incorporated in each process as a protected procedure. The text of the program hardly had to be changed at all, and the changes which were made were trivial. Partly as a result of this observation, a programming style was adopted for the rest of the operating system with a view to facilitating similar rearrangements later [8]. This turned out to be strikingly similar to the abstract models of this paper.

The ease of rearrangement of the CAP system was a consequence of a programming style originally adopted because of the protection structure of the machine, but there seems little reason to doubt that similar conventions could be adopted without loss in a more ordinary computer. B. J. Stroustrup has made some proposals in [9] which are relevant here.

Underlying Differences between Styles

The conclusion which we can draw from our canonical models and the observations we made about them is that there is no inherent difference between the two styles of system design or the programs that use them. That is, the two styles lead to client systems with similar program structure and performance (a "zeroth-order" consideration). Furthermore, the computational complexity of the implementations of the system facilities to provide the two styles is similar (a "first-order" consideration). Thus the basis for preferring one style to the other must be found in some second- or higher-order consideration. It must be two or more steps removed from the primary consideration of the designer of the application of the system. We believe that this basis is in the nature of the substrate -- i.e., machine architecture and/or programming environment -- on which the process and synchronization facilities are implemented. The factors and design decisions of the system upon which the process and synchronization facilities are built are the things which make one or the other style more attractive or more tedious.

Thus on one machine, the notion of process may be intimately tied to that of virtual memory, and it may be easy to allocate message blocks and queue messages, but very difficult to to build a protected procedure call mechanism. In this case, a message-oriented style is probably the best. On another system with an Algol-like stack and block structured allocation, the procedure-oriented model is probably more suitable. Other such factors include the organization of real and virtual memory, the size of the stateword which must be saved on every context switch, the ease with which scheduling and dispatching can be done, the arrangement of peripheral devices and interrupts, and the architecture of the instruction set and the programmable registers. These are usually chosen or constrained before the design of any application (or even the operating system kernel) is contemplated. It is rare that the operating system designer has a great influence on these factors and even rarer that he has complete control over them. Thus he normally faced with having to build the most reasonable set of primitives he can without incurring large computational penalties in the most basic operations of his system.

If this is the case, then the arguments about processes and synchronization which are often found in the corridors of organizations actively designing a new system (and which occasionally find their way into the literature) take on a decidedly non-technical tone. In our experience, they tend to be highly emotional, they consume far more energy than any other part of the system, they occasionally lead to organizational difficulties, and they are about issues which this analysis suggests are irrelevant. They are characterised by someone representing one style as being unable to reconcile the system organization postulated by the other style with the limitations and constraints he has learned by analysis or bitter experience, and by a mutual feeling on the other side. Part of the problem is that the the common notion of process evokes wildly different implications in the two worlds, which we have shown is indeed the case. The result of such arguments is rarely an understanding of the equivalence of the two approaches, but rather an

unpleasant compromise for all involved.

Conclusions

Our analysis is likely to be controversial. Like those of any empirical science, our conclusions cannot be accepted without a lot of thought and supporting experiments. We have found that the duality between the two categories of system design is a notion which defies belief amongst a non-trivial sample of our colleagues. The observations about the similarity of program logic, code, and performance are particularly hard to accept when the universe of discourse is not one of naturally occurring objects but man-made ones. Time will tell whether they are correct.

We have several objectives in developing this analysis. First, we want to eliminate some of the uninformed controversy about which kinds of systems are "better" to build. We find merit in both styles, with respect to structure, performance, logical soundness, elegance, and "correctness." Second, we are able to eliminate several degrees of freedom in the design process, thus allowing the design of better, more consistent, more reliable systems at lower cost. Once a choice is made between the two styles, many of the properties of "good" system design follow naturally. It is no longer necessary to make separate, independent choices about related issues, with the risk of introducing some fundamental incompatibility which will not be perceived until too late after the system starts to operate. Finally, we remark that the equivalence between the two styles of system design suggests that it might be possible to devise a uniform way of modelling the interactions between system components, whether by messages or by procedures, in order to derive better means of calculating the system performance before it is designed instead of after.

Acknowledgements

We are indebted to Michael Melliar-Smith for many fruitful discussions on this subject and, in particular, for his observation that the application programs for two dual systems are, in general, identical. We are also grateful to our colleagues at the Xerox Palo Alto Research Center and System Development Division for their long, loud, and often emotional and heated debates on what kind of process mechanism to include in the Mesa language and system -- debates which created the necessity to perform this analysis and show the equivalence of the two approaches, thereby permitting a purely technical resolution of that issue.

References

1. IBM Corporation, *Operating System/360: Concepts and Facilites*, Poughkeepsie, New York.

2. General Electric Company (Marconi-Elliot Division), Borehamwood, London, Great Britain.

3. W. Wulf, R. Levin, and C. Pierson, "Overview of the Hydra Operating System Development," *Proceedings of the Fifth Symposium on Operating Systems Principals*, Austin, Texas, November, 1975.

4. D. M. England, "Capability concept mechanism and structure in System 250," *Proceedings of the International Workshop on Protection in Operating Systems*, IRIA, Rocquencourt, France, August, 1974.

5. C. M. Geschke, J. H. Morris, and E. H. Satterthwaite, "Early experience with Mesa," *Proceedings of ACM Conference on Language Design for Reliable Software*, Raleigh, North Carolina, March 1977.

6. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, 17, 10, pp. 549-557, October 1974.

7. R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and it protection system," *Proceedings of the Sixth Symposium on Operating System Principles*, Purdue University, Lafayette, Indiana, November 1977.

8. R. M. Needham, "The CAP project - interim evaluation," *Proceedings of the Sixth Symposium on Operating System Principles*, Purdue University, Lafayette, Indiana, November 1977.

9. B. J. Stroustrup, "On unifying module interfaces," *Operating System Review*, 12, 1, pp. 90-98, January 1978.