AN ABSTRACT OF THE THESIS OF

Mark Allen James for the degree of Master of Science in Computer Science presented on April 13, 1983
Title: Duality and the Bounded-Buffer Problem

Abstract approved: ________ Dr. Ted G. Lewis

The paper, "On the Duality of Operating System Structures," by Lauer and Needham[21], claims that operating systems can be modeled as procedure-oriented or message-oriented, and that the two models are duals of each other. Duality, in this case, means that the models are logically and functionally equivalent, and have the same run-time performance. The duality paper concludes that, since the models are equivalent, the choice between models is made based on the ease of implementation of the primitives, or the fit of the primitives to the underlying machine, not on the application being solved.

An experiment is presented which derives dual solutions to the bounded-buffer problem and then analyzes the solutions with respect to the duality claims. The majority of the duality claims are not supported by this thesis, therefore, the conclusion of the duality paper is rejected.
The first claim is rejected because of the bias towards the procedure-oriented model. To support this claim an example purposely biased towards the message model is presented. Performance equivalence is rejected because of the semantic difference between monitor exiting, in the procedure model, and the manager process immediately looking for the next request, in the message model[22]. A semantic alternative is presented which corrects the semantic difference, however, performance will still vary between models even with the semantic "patch." The difference being that the monitor inherits its priority from the calling procedure, whereas, the manager process is a stand alone, high priority process.

The conclusion that the application should not influence the model selection is rejected, and an alternate criteria for model selection is proposed. Both biased models are too restrictive to serve as a "general purpose" programming style. If the application lends itself to a particular model, then use that model. If performance is a requirement (we show performance variance between the two models), than select the model that gives the required performance. If reliability is an issue, it may dictate a closely coupled program structure, making one model more attractive than the other.
Duality and the Bounded-Buffer Problem

by

Mark Allen James

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the degree of
Master of Science

Completed April 13, 1983
Commencement June 1984
ACKNOWLEDGEMENTS

Ted Lewis, Oregon State University.
Fred Tonge, Oregon State University.
Terry Hamm, Tektronix, Inc.
Tektronix, Inc., for the use of the DEC System 20.
Intel Corporation, for the use of an MDS 800.
Lorine and Jessica for making it possible.
TABLE OF CONTENTS

I. Introduction
   A. Statement of the Problem 1
   B. Hypothesis and Experiment 2

II. Inter-Process Communication (IPC)
   A. What is IPC? 3
   B. Approaches to IPC 4
   C. Messages 5
   D. Procedures 8

III. The Duality Paper
   A. Summary 10
   B. Models 10
   C. Resource Manager Example 12
   D. Primitive Mapping 15
   E. Claims 16
   F. Problems 17

IV. The Experiment
   A. The Bounded-Buffer Problem 22
   B. Procedure-Oriented (Monitor) Solution 24
   C. Message-Oriented Solution 26
   D. Execution Sequences 28

V. Analysis and Conclusions 33

VI. Future 37

VII. Bibliography 38

VIII. Appendices
   Appendix 1. More Producer/Consumer Examples 42
   Appendix 2. Performance Measurement 44
   Appendix 3. Monitor Optimization 45
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Procedure-oriented v.s. message-oriented</td>
<td>7</td>
</tr>
<tr>
<td>2.</td>
<td>Procedure-oriented execution sequence</td>
<td>29</td>
</tr>
<tr>
<td>3.</td>
<td>Message-oriented execution sequence</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>Message-biased procedure example</td>
<td>31</td>
</tr>
<tr>
<td>5.</td>
<td>Message-biased message example</td>
<td>32</td>
</tr>
</tbody>
</table>
Duality and the Bounded-Buffer Problem

I. Introduction

A. Statement of Problem

In 1978, a controversial paper "On the Duality of Operating System Structures," by Lauer and Needham[21], modeled operating systems as either procedure-oriented or message-oriented, and claimed that the two models are duals of each other. Duality, in this case, means that the models are logically and functionally equivalent, and perform the same. The duality paper concludes that, since the models are equivalent, the choice between models is made based on the ease of implementation of the primitives, or the fit of the primitives to the underlying machine, not on the application being solved. The duality paper has sparked discussion[32,33] and research[20,29,35] into the duality claim.

Are the duality claims supported when we solve the bounded-buffer problem? In this thesis we attempt to answer this question.
B. Hypothesis and Experiment

Our major result is that when we solve the bounded-buffer problem not all of the duality claims are supported, therefore, the conclusion is not supported.

The experiment presents dual solutions for the bounded-buffer problem following the "strict programming style" required by the duality paper. A dual solution is derived by substituting message-oriented primitives for Hoare's procedure-oriented solution to the bounded-buffer problem[12]. Performance equivalence is examined by presenting a comparison of execution sequences for the bounded-buffer problem for both models. The execution sequences point out a mechanism difference in the procedure-oriented model which precludes performance equivalence. In addition, the strict programming style required by the duality paper, biases the solution towards the procedure-oriented model instead of presenting a general programming style, which is equally applicable to both models, as the duality paper claims. Therefore, since all of the duality claims are not supported by the experiment, we reject the conclusion of the duality paper and propose an alternate criteria for model selection.
II. Inter-Process Communication (IPC)

A. What is IPC?

Before examining duality we need to understand inter-process communication. Operating systems exist to provide common services to a program, or set of programs, and to make possible sharing of hardware resources[1]. Programs are made up of one or more processes. These processes can be supported explicitly in the language of the program, for example ADA[15] and MESA[20], invoked by a procedure "call"[25,30], or even triggered by an "escape" sequence intercepted by special purpose hardware, such as a math coprocessor (e.g. Intel 8087). When more than one process is executing simultaneously on a single C.P.U., we call them concurrent processes[4]. If these concurrent processes need to synchronize, this can be done using locks, or P and V operations [8,31]. If these processes need to communicate, the portions of code that manipulate the shared data are called critical regions[8]. These programmed operations on shared data must exclude one another in time. This act of communication is referred to as interprocess communication, or IPC. Locks, P and V operations, monitors[12], and messages can all be used for IPC mechanisms.
B. Approaches to IPC

There are roughly three approaches to IPC[32]: master/slave, dialogue, and mailing.

Master/Slave - master always in control, slave always available; slave usually notifies completion; characterized in software by procedure call, in hardware by control lines; tightly coupled, efficient, no need for messages.

Dialogue - must acquire permission, carried out asynchronously, temporary connection; procedure calls are adequate if on same host, else messages can be used, or global data.

Mailing - implies some type of buffering as information is not sent directly, but to an intermediate stop; receivers and senders operate totally asynchronously because of the buffering; procedure calls will not support mailing without user supplied buffering; messages are required.

Co-Master - control is exchanged synchronously, for example coroutines[37].

Procedure-oriented systems could be roughly classified as a master/slave approach because the "call," to enter a procedure, will usually not return until the procedure has completed. Messages exemplify the mailing approach because messages were developed to implement mailing systems. Before examining duality in detail, we will clearly define procedure-oriented and message-oriented mechanisms and identify the differences.
C. Messages

Discussing exact syntax and semantics for message-oriented systems is more difficult than for procedure-oriented systems because there are no standards for messages, and the terminology is not widely understood[32]. There are many specific examples of message semantics in the programming language area: "Distributed Processes[9]," *MOD[3], CSP[13], Smalltalk[16], PCL[23], PLITS[6], Actors[11], and ADA[15]. Operating system examples include: TRIX[36], Mininet[25], Pilot[28], HXDP[17], Staros[18], Medusa[27], Suppose[2], and MICROS[38].

Messages contain information which is transferred from one process to another, similar to a parameter mechanism[6]. Unlike tightly coupled parameter passing mechanisms, messages are loosely coupled IPC mechanisms. Sometimes when a process "sends" a message, it doesn't care if it is received. In essence, the message has an identity, or existence, of its own and may outlive the creating process. Messages imply buffering, so we classify all buffering schemes as message systems [29].

The duality paper defines the message-oriented model as a static set of processes communicating by passing messages, which are queued at their destination until another process can act upon them.

Strictly speaking, messages are shared by copying them
from one environment to another, for example pass-by-value (see figure 1). In actual practice many message-oriented systems are implemented by passing pointers to the data, pass-by-reference: RMX/80[30], MBOS[26], Mininet[25], and PILOT[28]; but a few systems religiously copy shared data from one environment to another, pass-by-value: NXDP[17].
Figure 1. Procedure-Oriented v.s. Message-Oriented
D. Procedures

The procedure call is well understood and widely used (see figure 1). In procedure-oriented systems data is shared globally, employing some form of memory interlock. This insures that accesses to global data exclude one another in time. Examples in the area of operating systems include HYDRA[39], and the Plessey System 250[5].

The duality paper defines the procedure-oriented model to assume a very dynamic set of processes, being rapidly created and deleted, which communicate by procedure calls and shared data; the data is protected by some form of memory interlock, for example a monitor. A monitor contains shared data and procedures that access the shared data. Lewis defines a monitor abstract data type as follows[24]:

```
monitor
  <synchronization conditions>
  <permanent data>
  <access procedures>(i.e. entry procedures)
  <initialization code>
```

The programming language Concurrent Pascal[10] has incorporated monitors into its syntax, and the MESA systems programming language[20] has been enhanced with a monitor facility.

In the procedure-oriented model IPC occurs by procedure calls and shared, global data. Synchronization can be achieved using monitors with condition variables.
Procedure calls can be modeled with messages, but messages can be implemented with procedure calls only when the user supplies the buffering mechanism. In practice, messages mechanisms are often invoked with procedure calls[26,30]. Deviating from this strict categorization, the ADA "task" primitive does not distinguish between message-oriented and procedure-oriented[15].
III. The Duality Paper

A. Summary

Using a resource manager as an example, the duality paper defines two operating system models, their associated primitives, and then asserts that the models are equivalent. For each primitive in one model there is an equivalent primitive, or set of primitives in the other model. Also, the claim is made that the equivalence of the primitives includes performance equivalence. That is, one set of primitives does not outperform the other set. Unfortunately, the duality paper does not present execution results for the two models so that they can be compared; it does not reveal execution times required by the primitives presented; and it overlooks a difference in the primitives which prevents performance equivalence from being achieved. Nevertheless, the duality paper does present a criteria for classification and comparison of operating systems, and asserts that the choice between models not be based on the application being solved, but on the fit of the model to the underlying machine.

B. Models

The message-based model can be summarized as follows:
* Messages and message identifiers. The message is the unit of communication exchange. Message identifiers provide a way to refer to a specific message.

* Message channels and message ports. A message channel identifies the destination of a message. A message port is a queue for holding messages. A message channel must be bound to a port, but a port can have many channels bound to it.

* Processes. Processes are active entities with behaviors. A process defines some message ports on which it receives messages, and it sends messages to message channels.

* Operations:
  - Create Process. Creates an instance of a process and binds the message channels it references to existing message ports.
  - Send Message [message CHANNEL, message BODY] returns [message ID]. Creates a new message which is sent to the message CHANNEL. The message ID is used for referring to the new message and identifying replies to it.
  - AwaitReply [message ID] returns [message BODY]. Causes the process to wait for a reply to a specific message.
  - WaitForMessage [set of message ports] returns [message BODY, message ID, message port]. Causes the process to wait for a message on any of the set of message ports. The message ID and message port are returned with the message body for future reference and so the process knows from which port the message came.
  - SendReply [message ID, message BODY]. Sends a reply message to a particular message.

The procedure-based model can be described as follows:

* Procedures. Procedures are active entities with behaviors. A procedure is always associated with a module that defines its scope and the global data available to it.

* Modules and monitors. A module is a collection of procedures and data. A monitor is a special module with a lock associated with it to prevent more than one of its procedures from executing at a time.

* Condition variables. Condition variables are part of the
monitor mechanism, providing more flexible synchronization than the monitor mutual exclusion lock. A condition variable is contained within a monitor and has associated with it a queue of processes.

- Synchronous call. The traditional procedure call mechanism, similar to that in Algol or Pascal. The calling process waits for the termination of the procedure before proceeding. The form of the operation is

Procedure Name[parameter list] returns[result list].

- Asynchronous procedure call. A modification to the definition of procedure call that permits the calling process and the procedure to proceed concurrently. It is provided by the following operations:

-- Fork procedure name[parameter list] returns [process ID]. Starts the procedure executing as a new process. The process ID is used to identify the new process.

-- Join[process ID] returns[result list]. The process issuing this operation waits until the process identified by the process ID terminates. The results are returned as if from a synchronous procedure call. The Joined process is destroyed.

- Wait[condition variable]. The process issuing this operation releases the monitor lock, suspends execution and waits on the queue associated with the condition variable.

- Signal[condition variable]. A process that is waiting on the condition variable's queue is removed from the queue and permitted to resume execution as soon as it can recover the monitor lock.

C. Resource Manager Example

The following message-oriented solution, a resource manager, is presented in the duality paper to represent the "standard programming style" that is essential to the duality argument. This style allows the substitution of
primitives between models necessary for the duality mapping.

BEGIN m: message BODY;
  i: port ID;
  s: SET OF port ID;
  ... -- local data and state information
  -- for this process
initialize;
DO FOREVER;
  [m,i,p] <- WaitForMessage[s];
  CASE p OF
    port1 => ...
    port2 => ...
      IF resourceExhausted THEN
        s <- s - port2;
        SendReply[i,reply];
    ...
    portk => ...
    s <- s + port2
  ...
  END_CASE;
ENDLOOP;
END.
A "standard" resource manager is defined for the procedure-oriented model as follows:

```
ResourceManager: MONITOR =
c: CONDITION;
resourceExhausted: BOOLEAN;
. . . -- global & state info. for this process

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

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

initialize;

END.
```

These two examples illustrate different mechanisms for process synchronization and communication. The correspondence of primitives is the key to the duality claim.
D. Primitive Mapping

Given the structure of the resource manager, the duality paper claims the following mapping between the primitives of the two models:

**Message-oriented**

- Processes, CREATEPROCESS
- Message Channels
- Message Ports
- SENDMESSAGE; AWAITREPLY (immediate)
- SENDMESSAGE; ... AWAITREPLY (delayed)
- SENDREPLY
- main loop of std. resource manager, WAITFORMESSAGE stmt.,
- CASE statement
- arms of CASE statement
- selective waiting for messages

**Procedure-oriented**

- Monitors, NEW/START
- Ext. Proc. identifiers
- ENTRY proc. identifiers
- simple procedure call
- FORK; ... JOIN
- RETURN (from procedure)
- monitor lock, ENTRY attr.
- ENTRY procedure decl.
- condition variable, WAIT, SIGNAL

Each primitive in a particular model has a corresponding primitive, or set of primitives, in the other model that perform the identical function. For example, a SENDMESSAGE operation followed by an AWAITREPLY, later, is used in the message-oriented model, but a corresponding FORK, JOIN pair would accomplish the same thing in the procedure-oriented model. A more interesting correspondence exists between the processes of the message model and the monitor procedures of the procedure model. The choices in the CASE statement of the message model resource manager correspond to entry procedures in the monitor of the procedure model; the main loop
WAITFORMESSAGE and cases of the message model serve the same function as the monitor lock and entry procedures of the procedure model, that is mutually exclusive access to services. The case statement sorts out the service requests as do the entry procedure names contained in the monitor.

One cannot select primitives from both columns and build a combination solution because the mechanisms are not divisible. However, in some cases a subsystem could be constructed using one model and then combined with another subsystem employing the other model. The duality paper states that "attempts to combine fundamental characteristics from the two categories have met with failure or have been abandoned." If this programming style is strictly followed then a program can be easily transformed from one model to the other. This is the heart of the duality claim.

E. Duality Claim

The duality claim states:

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 duality paper concludes, since the models are equivalent, one would select a particular model over the other because the primitives of that model more closely match the underlying hardware, or because one set of primitives is easier to implement, given the underlying machine architecture.

F. Problems

Functionally the models are equivalent. There are applications implemented on both procedure-oriented and message-oriented systems, which use the same algorithms and perform the same functions. But the duality claim further states that the implementations can be made identical, differing only in the primitives used and non-essential details. This textual similarity of the solutions raises the question of whether the solutions are inherently, textually similar or whether the resource manager was carefully chosen because the two solutions can be made textually similar[29]. We will show that the resource manager example is biased towards the procedure-oriented
model. Proclaiming textual similarity is very subjective and difficult to measure.

The performance equivalence claim comes from the idea that if there is a one-to-one correspondence between the primitives, functionally, then it follows that they will perform the same, given optimal implementations of both. This claim is presented very informally and with no supporting data. Example implementations of the primitives are not revealed, and execution times comparing the primitives are not presented. This is essential to prove performance equivalence. The duality paper further states that optimizing for performance gains in one model can be matched by optimizations in the other model; that is, one model cannot be made more efficient than the other.

In addition, a strong assertion is made that events such as: enqueuing, dequeuing, process scheduling, and process switches should happen in the same order in both models. Reid claims that the duality paper does not demonstrate duality, but reveals a macro interface to a communication construct that can be implemented in either model by using a particular set of primitives[29]. There is a difference in the IPC mechanisms of the two models that precludes performance equivalence. The duality claim states that substituting one set of primitives for the other does not alter the algorithm of the program, in fact preserves performance because "corresponding events will
happen in the same order." The difference discovered is at the level of communication mechanism. In the procedure-oriented model, the monitor entry procedure inherits its priority from the "calling" procedure and will return to it when exiting the monitor. In the message-oriented model, the manager process is a high priority process that will immediately execute the next request when finished with the current request. This will cause a difference in the execution sequence between models which inhibits performance equivalence[22]. It will cause process switches to occur at different execution points in the two models, therefore affecting performance.

Several problems arise when one tries to take the primitives of the duality paper and verify the claim that one set of primitives can be substituted for the other. Valid and invalid uses of the primitives are not specified, and too much of the syntax and semantics of monitors is left to the intuition of the reader[29]. The following questions arise concerning the message-based primitives:

* Can message/reply pairs be interleaved?
* Can message/reply pairs be nested?
* Can messages be forwarded?

The following are monitor issues:

* When signalled, a waiting procedure does not get priority when attempting to reclaim the monitor lock.

* Using return statements inside monitor entry procedures:
  - omitted from Hoare's discussion
  - placement in entry procedure body
  - semantics of return in entry procedure
Confining conditional waits to monitor entry/exit improves structuring.

Hoare was very explicit about the effect of signalling from a monitor procedure, "we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program." Originally it was suggested that FIFO logic be used to select the procedure to be allowed to reenter the monitor after a signal, but Hoare acknowledged the need for priorities by allowing a priority specification with the conditional wait. Thus, the waiting procedure with the greatest need resumes execution first. The duality paper deviates from Hoare's monitor approach by requiring a signalled procedure to enter the ranks of those procedures waiting to enter the monitor, with no high priority treatment.

Since no return statements are used in Hoare's examples, a number of questions concerning their usage arise. One can only make subjective judgements concerning their usage because of the lack of published information. We will assume the return was meant to terminate the entry procedure.

As for the placement of conditional waits, Hoare did not restrict their placement. The duality paper suggests it might be used in a more "structured fashion" if
restricted to entry, exit points in an entry procedure. It is likened to the unstructured "GO TO" because of the difficulty involved in preserving the monitor invariant, given that some of the local data of the monitor may have been modified by the monitor procedure.

Too much of the semantics of the monitor construct is left to the previous experience, or intuition, of the reader. In order to proceed from where the duality paper left off, one must attempt to come up with a monitor model that will perform identically to the message model described. This is a difficult, if not impossible, task. Much work has been done on improving the performance of the monitor construct by changing the semantics of it (see Appendix III). We will look at the bounded-buffer problem and present dual solutions. Then we will examine the solutions to analyze the duality claims.
IV. The Experiment

A. Bounded-Buffer Problem

The bounded-buffer problem[12] is a representation of an abstract idea which deals with a sequence of "portions." Two programs (processes), running asynchronously, access the sequence of portions; one process, the "producer," appends a new portion at the end of the sequence, and another process, the "consumer," removes a portion from the beginning of the sequence. Initially the sequence is empty. Two operations, append and remove, are needed to manage the sequence. Append is repetitively invoked by the producer process to concatenate a portion to the end of the sequence, and remove is repetitively invoked by the consumer to return a portion from the beginning of the sequence. The bounded-buffer problem is actually a queue manager because of its FIFO handling of portions. Either message-oriented or procedure-oriented primitives can be used to solve the bounded-buffer problem.

A procedure-oriented, monitor solution encapsulates the queue, the append procedure, and the remove procedure into a monitor. Producer and consumer procedures make "calls" to entry procedures, append and remove, to add to and remove portions from the queue.

A message-oriented solution consists of a producer
process creating messages, which contain a portion to add to the sequence, and then "sending" the message to a message channel, that has an associated message port, which is the queue. The consumer process "waits" for messages, which contain a portion that is removed from the sequence.
B. Procedure-Oriented(Monitor) Solution

Hoare published a procedure-oriented solution to the bounded-buffer problem[12]. The bounded-buffer monitor was provided, however the producer and consumer procedures were left to the reader.

The following is a procedure-oriented asynchronous producer procedure (i.e. with FORK only, producer doesn't care if data received):

```plaintext
producer: PROCEDURE;
BEGIN p: portion;
    pid: process ID;
    DO FOREVER;
        NOTE produce a portion p;
        pid := FORK append[p];
        NOTE join is omitted because don't care when append completes;
    END DO;
END producer.
```

A procedure-oriented synchronous consumer

```plaintext
consumer: PROCEDURE;
BEGIN p: portion;
    DO FOREVER;
        p := remove;
        NOTE consume the portion;
    END DO;
END consumer.
```
Hoare's procedure-oriented bounded-buffer monitor

bounded_buffer: MONITOR;

BEGIN buffer: ARRAY 0..N-1 OF portion;
    lastpointer: 0..N-1;
    count: 0..N;
    nonempty, nonfull: condition;

append: ENTRY PROCEDURE (x: portion);

BEGIN IF count = N THEN nonfull.wait;
    NOTE 0 <= count < N;
    buffer[lastpointer] := x;
    lastpointer := lastpointer + 1;
    count := count + 1;
    nonempty.signal;
END append.

remove: ENTRY PROCEDURE (RESULT x:portion);

BEGIN IF count = 0 THEN nonempty.wait;
    NOTE 0 < count < N;
    x := buffer[lastpointer - count];
    count := count - 1;
    nonfull.signal;
END remove.

    count, lastpointer := 0; NOTE initialization;

END bounded_buffer.
C. Message-Oriented Solution

The following is a message-oriented solution to the bounded-buffer problem.

A message-oriented asynchronous producer process

producer: PROCESS;

BEGIN c: messageCHANNEL;
   p: portion;
   i: messageID;
   m: messageBODY;

   DO FOREVER;
      NOTE produce portion;
      i := SendMessage[];
   END DO;

END producer;

A message-oriented synchronous consumer process

consumer: PROCESS;

BEGIN c: messageCHANNEL;
   i: messageID;
   m: messageBODY;

   DO FOREVER;
      NOTE create messageBODY m;
      i := SendMessage[c,m];
      m := AwaitReply[i];
      NOTE consume portion m.p;
   END DO;

END consumer.
The following is the message-oriented manager process:

```almond
manager: PROCESS;
BEGIN  buffer: ARRAY 0..N-1 OF portion;
    lastpointer: 0..N-1;
    count: 0..N;
    mid: messageID;
    pid: portID;
    s: SET OF portID;
    msg: messageBODY;

    s ::= port1; NOTE initially only append allowed, nothing to remove;

    DO FOREVER;
        [msg,mid,pid] := WaitForMessage[s];
        CASE pid OF
          port1: BEGIN NOTE append requested;
              NOTE buffer has room,
              0 <= count < N;
              buffer[lastpointer] ::= msg.p;
              lastpointer ::= lastpointer + 1;
              count ::= count + 1;
              NOTE test for buffer full;
              IF count = N THEN
                  BEGIN NOTE buffer full;
                      NOTE buffer append requests;
                      s ::= s - port1;
                  END THEN;
              NOTE allow remove requests;
              s ::= s + port2;
            END port1;
          port2: BEGIN NOTE remove requested;
              NOTE there is a portion to remove,
              0 < count < N;
              msg.p := buffer[lastpointer-count];
              count ::= count - 1;
              NOTE test for buffer empty;
              IF count = 0 THEN
                  BEGIN NOTE buffer empty;
                      NOTE buffer remove requests;
                      s ::= s - port2;
                  END THEN;
              NOTE allow append requests;
              s ::= s + port1;
              CALL SendReply[mid,msg];
          END port2;
        END CASE;
    END DO FOREVER;
END manager.
```
D. Execution Sequences

The execution sequence for the procedure-oriented model (see figure 2) reveals an uncertainty as to what assumptions to make about exiting a monitor:

1. if we assume that upon exiting the monitor, the append process continues to execute (see figure 2, number 7), then the sequences between models will vary because in the message-oriented model the manager process always looks for the next request, immediately (see figure 3).

2. if we assume that upon exiting the monitor, the monitor append process is preempted (or dies), then alternatively the scheduler could run a process waiting to enter the monitor (see figure 2, note at bottom).

Typically, when we "call" or "FORK" to a monitor, the monitor inherits its priority from the invoking procedure, therefore it is unlikely that the next pending monitor request would be honored without first "returning" to, or "JOINing" the procedure invoking the monitor[22]. Figure 2, number 7, is assumed to be the correct choice. The result is a difference in the execution sequences between the models.
Explanation:

1. producer procedure creates portion Pn
2. producer FORKS a process, append, to call monitor append
3. producer continues executing asynchronously
4. process append tries to enter monitor
5. process append successfully enters monitor (assume buffer not full)
6. appends Pn to buffer
7. signals non-empty condition
8. exit monitor
9. process append continues (will "kill self")

Note - a semantic alternative to exiting the monitor is to check the monitor lock queue and if not empty, switch processes to run one of the processes waiting to enter the monitor

Figure 2. Procedure - Oriented Execution Sequence
Explanation:

1. producer process creates a portion Pn, a message Mn, and places Pn in Mn
2. producer Sends message Mn to port1 requesting an append operation
3. producer continues executing asynchronously
4. high priority manager process preempts producer process(already waiting for requests from port1)
5. executes Case statement pid of port1
6. add Pn to buffer (assume buffer not full)
   add port2 to S enabling remove operations (will not happen until WaitForMessage is executed)
   return to beginning of loop 4 wait for next request (append and remove now allowed)

Figure 3. Message - Oriented Execution Sequence
```
"asynchronous" Producer: PROCEDURE
DO FOREVER
    create portion Pn
    FORK Append(Pn)
END DO

"synchronous" Consumer: PROCEDURE
DO FOREVER
    Pn = CALL Remove
    consume Pn
END DO

Port: MONITOR
BEGIN
    1: LINKED LIST OF portion
    nonempty: CONDITION

    Append: ENTRY PROC(p: portion)
    BEGIN
        link p into 1
        nonempty.SIGNAL
    END Append

    Remove: ENTRY PROC(RESULT p: portion)
    BEGIN
        IF 1 notempty
        THEN BEGIN
            remove p from 1
            RETURN p
        END THEN
        ELSE nonempty.WAIT
    END Remove

    END Port

Explanation:

1. FORK creates a process that attempts to gain
   the monitor Lock, whereas CALL waits
   for monitor exit

2. gaining entry to the monitor and waiting/signalling
   a condition are separate, divisible mechanisms
```

*Figure 4. Message - Biased Procedure Example*
DO FOREVER
    create message Mn
    create portion Pn
    \text{i}=\text{SendMsg}(Mn)
    \text{..}
END DO

"asynchronous"Producer: PROCESS

DO FOREVER
    Mn=WaitForMsg
    \text{..}
    consume Pn from Mn
END DO

"synchronous"Consumer: PROCESS

Explanation:

1. buffering and mutual exclusion are provided by a single mechanism

2. producer consumer processes are very simplistic

Figure 5. Message - Biased Message Example
V. Analysis and Conclusions

The duality paper claims: functional correspondence of the primitives of both models, algorithmic equivalence of solutions to problems, and performance equivalence of the primitives of both models. Therefore, the duality paper concludes, the choice between models should be made based on the fit of the primitives to the underlying machine, not on the fit of the problem being solved to the primitives. In other words, one model is not inherently preferable over the other model.

Functional correspondence of primitives is supported by the experiment. A dual solution is derived by taking Hoare's monitor solution to the bounded-buffer problem and substituting the message-oriented primitives, following the strict programming style prescribed by the duality paper. Algorithmic equivalence is demonstrated, but rejected because the "strict programming style" required is biased towards the procedure-oriented model, and performance equivalence is rejected because of a difference in the communication mechanisms that precludes performance equivalence[22]. Because two out of three of the duality claims are rejected, the conclusion is also rejected and an alternate criteria for model selection is presented.

Algorithmic equivalence is rejected even though syntactically similar solutions are produced when we follow
the strict programming style required by the duality paper. The claim is rejected because the strict programming style is biased towards the procedure-oriented model. The message-oriented solution is structured to simulate the monitor mechanism, not define a general purpose programming style as claimed by the duality paper. Specifically, two separate mechanisms, the WaitForMessage and Case statement serialize and sort out accesses to the shared data, just as the monitor lock and entry procedures do in the procedure-oriented mechanism. The manager process is unnecessary in the message-oriented example because the message passing mechanism provides: buffering, serializes accesses to the shared data, and fulfills the role of the manager process, all in a single, indivisible mechanism. Therefore, the strict programming style required by the duality paper biases the message-oriented solution towards the procedure-oriented environment.

The preceding message-oriented solution to the bounded-buffer problem is a procedure-biased way to represent this particular problem. By following the strict style of the resource manager, we are forced into a bounded-buffer solution that would be virtually, unbounded if programmed differently. A more message-oriented way to represent this problem is to create a message port for each interdependent set of processes (see figures 4, 5). The port mechanism will provide the necessary buffering and mutually
exclusive access. The disadvantage of the manager process is that it forces a much tighter coupling on the processes, messages are inherently a very loosely coupled mechanism. The shared data buffer and count are not needed, they are inherent in the message queue mechanism. The message biased solution is no longer a "bounded" buffering solution, but a more general buffering solution, because the buffering capability of the message passing mechanism is not bounded by a static buffer size, it is limited by the number of messages that can exist simultaneously in the underlying machine, a memory consideration.

Performance equivalence is rejected because the experiment pointed out a difference in execution sequences between models, resulting from the monitor exit mechanism, and the inherited priority of the monitor entry procedure which is characteristic of procedure calls.

We conclude that both programming styles are too restrictive to be considered general purpose. The selection of a programming style should not precede selection of the model(primitives). When a model is selected, an appropriate programming style should also be chosen. This will avoid unnecessary overhead, for example, the unnecessary manager process of the message-oriented experiment. The solutions may not resemble each other syntactically, and may perform differently. Therefore, we reject the duality paper conclusion that the application
does not influence the choice of models. We propose an alternate criteria for model selection.

Several factors should be considered when choosing an operating system:

1. the application may lend itself to a particular model, for example, a network application that requires messages.

2. if performance is important to the application, for example, real-time applications, one model may perform better due to some special purpose hardware support.

3. structuring and reliability in an application may require a very close coupling (procedure-oriented), making one model more attractive than the other.
VI. Future

Even though the procedure call mechanism is more widely understood than the message passing mechanism, there is still a lack of standards for both the syntax and semantics of both mechanisms. In order for future work to be done in the area of performance equivalence, there must exist a complete description of the syntax and semantics of both sets of primitives, in order that common primitive functions can be identified and accounted for. Only then can the true cost of the primitives be compared using the same accounting system. Actual machine implementations are not useful, because what is important is not the actual execution speed of a primitive, but the accounting for common functions underlying the primitives, for example, context switches, and buffer allocations.
VII. Bibliography


[14] Howard, John H., "Signaling in Monitors," University of Texas at Austin.


[26] MBOS, Message-Based O.S. for 4100 Family Terminals, Tektronix, Inc.


VIII. Appendices
Appendix 1. More Producer/Consumer Examples

This appendix contains additional examples of producer and consumer procedures and processes not presented in the main portion of the thesis.

a procedure-oriented synchronous producer procedure

```
producer: PROCEDURE;
BEGIN p: portion;

DO FOREVER;
   NOTE produce a portion p;
   CALL append[p];
END DO;
END producer.
```

a procedure-oriented asynchronous producer procedure
(with FORK/JOIN pair)

```
producer: PROCEDURE;
BEGIN p: portion;
pid: process ID;
result: list;

DO FOREVER;
   NOTE produce a portion p;
   pid := FORK append[p];
   .
   .
   result := JOIN[pid];
END DO;
END producer.
```
a message-oriented synchronous producer process

producer: PROCESS;
BEGIN c: messageCHANNEL;
i: messageID;
m: messageBODY;
p: portion;

DO FOREVER;
    NOTE produce portion p;
    NOTE create message body m;
    m.p := p; NOTE put portion into messageBODY;
    i := SendMessage[c,m];
    m := AwaitReply[i];
END DO;
END producer.

a message-oriented asynchronous producer process
(simulates FORK/JOIN of the procedure-oriented model)

..
..
.. DO FOREVER;
    i := SendMessage[];
    ..
    ..
    m := AwaitReply[];
END DO;
..
Appendix 2. Performance Measurement

Preserving performance across the duality mapping requires that events happen in the same order, and that servicing those events costs the same in both models. The duality paper claims, without proof, that an optimization performed in one model has a dual optimization in the other model. Therefore, one set of primitives will not be more attractive because they execute faster. No performance data is presented in the duality paper, so performance equivalence is merely a conjecture.

Some performance data is available for procedure-oriented primitives as a result of the addition of monitors to the MESA systems programming language [20]. A unit of time called a "tick" is defined for measuring performance. One-fourth of the time needed to execute a \( \langle=\ b + c \) (i.e. two loads, one add, and one store) is defined to be a tick on that particular host machine. The MESA performance data follows:

<table>
<thead>
<tr>
<th>Construct</th>
<th>&quot;ticks&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple instruction</td>
<td>1</td>
</tr>
<tr>
<td>Call + Return</td>
<td>30</td>
</tr>
<tr>
<td>monitor Call + Return</td>
<td>50</td>
</tr>
<tr>
<td>process switch</td>
<td>60</td>
</tr>
<tr>
<td>Wait(conditional)</td>
<td>15</td>
</tr>
<tr>
<td>Signal(no one waiting)</td>
<td>4</td>
</tr>
<tr>
<td>Signal(process waiting)</td>
<td>9</td>
</tr>
<tr>
<td>Fork + Join</td>
<td>1100</td>
</tr>
</tbody>
</table>

FORK and JOIN were implemented in software rather than microcode because of their infrequency of use.
Appendix 3. Monitor Optimizations

A lot of effort has gone into optimizing the monitor construct since its conception. Kessels proposes a modification to Hoare's event queues for synchronization in monitors[19], because, as Hoare states, "it is much more difficult to be confident about the condition concept as a synchronizing primitive." The modified conditional variables are declared at the monitor level, together with a Boolean expression, but are not accessible monitor entry procedures. Whenever an entry procedure executes a WAIT, the corresponding Boolean expression is evaluated, then a decision is made whether to continue or suspend. Whenever a procedure is suspended, condition queues are checked first to determine which procedure to run next, before checking procedures waiting to enter the monitor. Stroustrup[34] proposes that a "long return" is especially beneficial in procedure-based systems. That is, if procedure A Calls procedure B, which in turn Calls procedure C, then when C returns to B, if all B does is return to A, then why not have C return to A. A performance gain of two percent can be realized especially in systems with little hardware support. The net effect of long returns on monitors is to release monitor locks earlier, which means fewer procedures waiting on monitor locks. Ford[7] proposes a generalized critical region
construct that requires less evaluation of complex waiting conditions and less context switches. Howard[14] suggests there are at least four methods of handling signaling in monitors:

- **SU** - signal and urgent wait (get priority over procedures entering monitor)
- **SR** - signal and return (why wait just to return?)
- **AS** - automatic signaling (whenever monitor entry procedure returns condition queues are evaluated; no signal)
- **SC** - signal and continue (keep going)

Note: Kessels' proposal is a combination of SU + AS.