THE DESIGN OF PARALLEL SYSTEMS:
AN APPLICATION AND EVALUATION OF MODULA

Gregory R. Andrews

TR 78-330
January 1978

Computer Science Department
Cornell University
Ithaca, New York 14853
THE DESIGN OF PARALLEL SYSTEMS:
AN APPLICATION AND EVALUATION OF MODULA

Gregory R. Andrews
Department of Computer Science
Cornell University
Ithaca, New York 14853

Modula is a new programming language for implementing dedicated, parallel systems. Following a systematic design technique, this paper illustrates the use of Modula for the design of a message switching communication system. A message switching system poses a number of interesting problems: a high degree of concurrent activity exists, a variety of IO devices need to be controlled, messages can have multiple destinations, and messages can be pre-empted. The strengths and weaknesses of Modula with respect to these specific problems and its utility as a general purpose language are evaluated.

Key Words and Phrases: structured multiprogramming, concurrent systems, Modula, message switching, software design, processes, monitors, modular design.

CR Categories: 3.81, 4.2, 4.21, 4.3

This work was supported in part by Battelle Columbus Laboratories under Scientific Services Program contract 0562 and in part by NSF grant MCS 77-07554.
1.0 Introduction

The unmistakable trend in recent years has been toward the use of high-level languages for systems programming. In an effort to improve upon available tools, three new languages have recently been designed: Concurrent Pascal [2] and Modula [8] aid in the design and implementation of multiprogramming systems while Euclid [4] is intended for implementing verifiable systems such as compilers or operating system nuclei. All three borrow heavily from the work of Wirth in the design of Pascal [7]. Although intended primarily for the development of small operating systems, both Concurrent Pascal and Modula are applicable to parallel systems in general.

In this paper, the design of one specific example, a message switching communications system, is developed using Modula. Our purposes are: (1) to present a system design technique; (2) to illustrate the use of Modula as a design, documentation, and implementation language; and (3) to evaluate Modula’s utility. A message switching communication system was chosen as the application because it shows the range of Modula’s applicability and presents a number of interesting implementation problems. Modula was chosen as the target language because it is specifically intended for dedicated multiprogramming systems, provides much needed facilities for controlling input/output, and appears to be very efficiently implemented [9,10].

The specific design technique used here is described in Section 2. The communication system itself is then developed in Sections 3-5. Section 3 specifies the functions and external interfaces of the system. Section 4 summarizes the major features of Modula and presents
the system organization, information and control flow, and block interfaces in terms of Modula components. Section 5 refines the organization by giving outlines of Modula programs for the most interesting parts of the system. Finally, Section 6 evaluates the utility of Modula for the design of parallel systems by reflecting on aspects of the communication system.

Modula is used here as a specification language since its compiler is not generally available. Even without a compiler though, we feel that programming any system in a structured, high-level language such as Modula is a valuable prelude to actual implementation. It serves as an intermediate step between specification and coding that helps one develop and reason about the implementation. It also provides meaningful documentation when used as comments in whatever implementation language is eventually employed.

2.0 Design Technique

The design described here was developed in three major steps: system specification, system organization, and program implementation. Each major step consists of a number of parts. The first step involves specifying the major functions of the system and the specific formats of input/output messages. This characterizes both what and how information is processed; hence it completely characterizes the user's view of the system. My role, as the designer, was to discuss system functions with intended users in order to understand and characterize the purpose and scope of the proposed system. In addition, it generally is helpful to lay out a representative hardware configuration in order to get a better feel for the size of the system.
The second step, system organization, involves successive refine-
ment of the system functions into a program structure in terms of
Modula constructs. Since Modula has processes and modules [8] as its
basic building blocks, various organizations in terms of processes
and modules were considered. First, important groups of processes
and modules, corresponding to groups of similar IO devices and to
major system functions, were identified. Second, each group was refiled
into a specific organization. Third, the interconnection of the
groups was specified, in this case in terms of Modula's interface
modules. Finally, the paths and order of information and control
flow through the system were traced. These ordered lists of actions
proved to be a convenient means for describing the organization to
others.

At this point the design was (and should be) discussed in detail
with the people contracting for the system. This allowed miscon-
ceptions and ambiguities about system functions to be clarified.
It helped the designer to be sure of the direction in which he was
headed and it helped the contractors to better understand the program
they would (hopefully) receive.

The final step is the implementation of each component. Imple-
mentation does not mean "start coding", however. First, the access
rights of each program component, in this case each process and module,
were specified. In particular, the procedures and parameters for each
interface module were precisely defined. Second, comments describing
the function of each procedure were written. Third, comment and code
outlines of each process were developed. Fourth, global data types
were defined. Finally each component was programmed by first de-
fining all variables and then writing language statements. Once the
organization was well understood and all variables were specified,
it became fairly easy to program each component. (This is not to
say that creativity was no longer needed!)

No design can proceed in a straight line through the above three
steps and the one described here did not. Some iteration will generally
occur since the steps are rarely independent. In this project, how-
ever, no change to one component affected anything more than the
parameters passed by components that invoked it.

3.0 Specification of a Message Switching Communication System

The message switching system considered here is modelled on
several communication systems currently employed by the US government
and NATO. It consists of a network of switching nodes connected via
trunk lines. Locally attached to each switching node are a number of
subscribers, an operator, archive tapes, and auxiliary memory. The
function of each node is to route input messages to one or more output
destinations. Input is received from local subscribers or from another
switching node (via a trunk line), stored on auxiliary memory and then
forwarded to output destinations, which can be either local subscribers
or other switching nodes. Because messages must be completely re-
ceived before being forwarded this type of communication is often
called store-and-forward message switching. A representative config-
uration, also indicating the type of hardware typically employed, is
shown in Figure 1. Since each switching node performs the same
functions, we will focus our attention on one node.
-5-

Figure 1
Representative System Configuration

subscriber: full duplex terminal
operator: full duplex terminal
switching node: processor(s)
archives: tapes
auxiliary memory: drum or disk
trunk: high speed communication lines(s)
Three successive phases are involved in processing each message: input, switch, and output. The input phase consists of reading input from a subscriber or trunk line and storing the message on both auxiliary memory and an archive tape. Each input message contains a header, body, and end marker as shown in Figure 2.

In the switch phase, the header is examined to determine the output destinations. For each destination, a directory is consulted to determine the appropriate output line to use (either local subscriber or trunk to a remote destination) and the message is queued for output. If a message is sent to more than one destination, each receives a copy.

In the output phase, a message is retrieved from auxiliary memory and written on the appropriate output line in the format shown in Figure 2. Each message contains a priority as part of its header so at all times the highest priority message for an output line is transmitted. This means that a partially written message can be pre-empted by a newly stored one. If pre-empted, a message is later re-transmitted in its entirety.

Each switching node has one operator who can send and receive messages like any subscriber. In addition, the operator monitors and controls the node's activity. The operator can request certain actions (e.g. cancel a message) and is notified when exceptions are detected (e.g. end of archive tape). Although necessary and important in an actual communication system, the operator will be ignored here because the main function of the switching node is to process input messages.
Figure 2
Subscriber and Trunk Message Formats

header: start of header code
message identifier (origin, date, and time)
size (omitted for subscriber output)
destinations (omitted for subscriber output)
priority
[other data not of concern here]
end of header code

body: sequence of characters (in 80 character blocks for trunk lines)

end: end of message code
In order to efficiently implement this communication system, four interesting problems need to be solved:

1. Maximal IO parallelism must be provided so that each subscriber and trunk is kept as busy as possible.

2. Two different types of IO devices, trunks and terminals, need to be controlled and yet both process similar messages.

3. The switch phase needs to coordinate the activity of all lines; in particular, output to multiple destinations must be coordinated.

4. An output message can be pre-empted, hence output controllers need to have a simple, efficient way to detect that pre-emption should occur; they also need to be able to restart transmission of a pre-empted message.

We now turn our attention to the design of a Modula implementation that solves each of these problems.

4.0 System Organization in Modula

For sequential programming, Modula contains a set of data types and statements based on those of Pascal. As building blocks for parallel programs, it also provides processes and modules. A process has the same structure as a procedure; namely, it has parameters, local variables, and a sequence of statements. It is also activated in the same manner as a procedure. The fundamental difference is that when a process is "called", the process and caller execute concurrently. Processes interact via global variables and modules.

Modules are like blocks in the ALGOL sense (they contain variables and statements). The difference is that a module forms a barrier between the objects it declares and those global to it. Its purpose
is to implement an abstraction such as a message queue. It does so by selectively defining those objects that represent an intended abstraction while hiding those objects involved in its representation. Two special types of modules play a key role for multiprogramming: interface modules and device modules. Interface modules, which correspond to monitors [2,3], guarantee mutual exclusion of defined procedures and provide for process synchronization via signal variables. Device modules are interface modules that contain device driver processes. There is one driver process in a Modula program for each addressable IO device. Each driver activates its associated device, waits for device completion via a special doIO statement, and interacts with other processes via device module variables.

4.1 System Components

Since there are three major phases involved in processing a message - input, switch, and output - it is natural that the design be organized into the major groups shown in Figure 3. The input groups control message input; the switch group coordinates activity; the output groups transmit messages over their lines. Since not all partially completed messages will generally fit in main memory, auxiliary memory is used as a "buffer" area for messages. Data therefore flows from input groups to auxiliary memory files and then to output groups. Control flows from input to switch to output. In addition, all message data and system actions are archived on tapes as shown.

We now refine this global organization by defining the specific
Figure 3
System Components

Input Groups
- create, write files
- control
- message data

Auxiliary Memory Group
- read files
- destroy files

Switch Group
- control
- system actions

Output Groups

Archive Group
actions and internal organization of each group. Because input and output groups both control IO lines, we consider their organization together then consider auxiliary memory and the switch. The archive is a simple addition.

Since subscriber and trunk lines are full duplex, we need to have both input and output driver processes for each line. Although they could be in separate device modules, aesthetically it makes sense to declare one device module having two internal drivers. Two major activities take place during input (output): formatting (or unblocking messages, and the actual reading (writing). The first is related to the application whereas the second is related to the hardware. Therefore they should be logically separated in case either changes. The final IO consideration is that within the switching node it is more efficient to transmit data a block rather than a character at a time.

Given these considerations, the chosen organization for input/output groups is shown in Figure 4. Each INCONTROL process reads characters, and parses them into blocks of a message. Each OUTCONTROL process does the reverse. Each SUBSCRIBER or TRUNK device module worries about the peculiarities of its device. Note that even though terminals and trunks are themselves quite different, the software organization of both groups is identical. Only internal details differ.

The auxiliary memory group provides temporary storage for messages. In particular, it provides INCONTROL and OUTCONTROL processes with sequential files. In addition, the auxiliary memory group must efficiently control the storage device. It therefore has two distinct levels: a file system level, which provides file operations and manages free space; and a device level, which schedules and performs IO
Figure 4
Input/Output Group Organization

One group for each subscriber terminal or trunk line

headers and data blocks

INCONTROL process

read

headers and data blocks

OUTCONTROL process

write

SUBSCRIBER or TRUNK
device module

input
driver

output
driver
and manages sector buffers. This leads to the organization shown in Figure 5. MEMORY defines five operations that provide an abstract file system. Within MEMORY there is the AUXMEM device module that implements the abstraction. Note that because AUXMEM is wholly inside MEMORY, no process global to MEMORY can directly access AUXMEM; global processes can only see and need only be aware of the file operations.

Since there is a need for one central controller, it makes sense that the switch group consists of one process, SWITCH, and several interface modules implementing various services. The function of SWITCH is to coordinate input and output; hence it needs to talk to every INCONTROL and OUTCONTROL process. (It also needs to talk to the operator; this is being ignored here.) In order to avoid the woeful inefficiency of polling, SWITCH should be signalled whenever there is work for it to do. Because it can only wait within one interface module at a time, SWITCH receives all its work from NOTICE. Each notice specifies a kind and, optionally, other parameters. The three major kinds of notices are "start of input", "end of input", and "done". A "start of input" notice causes SWITCH to receive a header from LINEINPUT, which it then stores in HEADERS; SWITCH then gives a REPLY back to the appropriate INCONTROL process. An "end of input" notice causes SWITCH to insert one output command in LINEOUTPUT for each output destination indicated in the header of the completed input message. The "done" notice signals completion of output to one destination; when all destinations are done, SWITCH deletes the saved header from HEADERS and destroys the MEMORY file containing the message. The interface between SWITCH and the INCONTROL and OUTCONTROL
Figure 5
Auxiliary Memory Group Organization
processes is shown in Figure 6.

4.2 Component Connections and Information Flow

We can now put the three main groups - subscriber/trunk, auxiliary memory, and switch - together. Figure 7 shows the order of actions taken in processing an entire message. The arrow on each arc indicates the direction of flow of parameters. Within each interface module signal variables are used to synchronize processes.

Although not shown in Figure 7, the operator communicates with SWITCH and MEMORY in the same manner as regular subscribers. Operator requests are sent to switch as special kinds of NOTICES and exceptions are sent from SWITCH to the operator via LINEOUTPUT. In a real application, the operator also communicates with a retrieval process in order to fetch old messages from archive tapes. These details are shown elsewhere [1].

5.0 System Implementation - Program Outlines

A Modula program is a module. In our case, the Switch module contains declarations of data types common to all components; device modules, interface modules and processes for each component in Figure 7; and code that initializes tables and activates the processes. Programs for the major components are outlined here in the order in which they were designed. We only show enough detail, however, to be able to meet our goals of illustrating and evaluating the use of Modula. Complete programs are given in [1].
Figure 6
SWITCH - INCONTROL/OUTCONTROL INTERFACE
Figure 7
Component Interfaces and Timing

INPUT PHASE
1. read header contents
2. send header
3. post "new input" notice
4. receive "new input" notice
5. receive header
6. store header
7. send reply
8. receive reply
9. create file and write header
10. read blocks and write to file
11. send "end of input" notice

SWITCH PHASE
12. receive "end of input" notice
13. update header
14. send output commands - one per destination

OUTPUT PHASE
15. retrieve header
16. receive output command
17. read file and write on line
18. call done operation
19. send "done" notice

TERMINATION
20. receive "done" notices - one per destination
21. delete header
22. destroy file
5.1 Global Data Types

The INCONTROL, OUTCONTROL, and SWITCH processes exchange several types of information. The two most important kinds are message headers and data blocks. To facilitate the declaration of variables referring to these types of information and to make use of Modula's type checking facilities, the format of header and block are specified in global type declarations. In particular, header is declared as¹:

```
  type header = record
    identifier : record
      origin, date, time : integer end;
    size : integer;
    outputcount : integer; (*number of destinations*)
    destinations : array 1 : "max" of integer;
  end
```

A block is simply an array of characters:

```
  type block = array 1 : "blocklength" of char;
```

Other types of variables for communicating with the archive module and operator are also needed in practice.

5.2 Device Modules

The SUBSCRIBER and TRUNK device modules provide interfaces to terminals and trunk lines, respectively. Each defines two procedures, read and write, that perform buffer management. Each also contains two driver processes, one for input and one for output, that perform

¹ As a coding convention, constants are denoted by names enclosed by quotes. They would in practice be declared as constants or given actual numerical values.
actual IO and synchronize with the procedures. An outline of a
SUBSCRIBER device module is shown in Figure 8. Character buffers
are used for IO; counter, pointers, and signals are used for synchron-
ization. Note that one SUBSCRIBER device module must be declared for
each terminal. There is no way in Modula to declare a type of device
module and subsequently declare as many instances as are required.

Each TRUNK module has the form shown in Figure 9. A TRUNK differs
from a SUBSCRIBER in that trunks transmit blocks instead of single
characters and numerous control characters are transmitted to synchro-
nize the trunk lines. Since trunks connect switching nodes, there
would be a TRUNK module in the node at each end of the line; informa-
tion output from one TRUNK module is read as input by the other.
The writecontrol procedure is used to output a control character.
Waitcontrol is used to wait for a control character response from prior
output of a data block. The response is sent by the switching node at
the other end of the trunk line; when received (via trunkread) its
presence and value are recorded via postcontrol.

5.3 Interface Modules

Six interface modules implement communication paths between the
processes: MEMORY, LINEINPUT, NOTICE, REPLY, HEADERS, and LINEOUTPUT.
MEMORY and LINEOUTPUT are both quite interesting; the others are
straightforward.

The MEMORY module defines five operations for managing files:
createf, writef, endwritef, readf, and destroyf. Its program is out-
lined in Figure 10. Createf is called by an INCONTROL process; it
device module SUBSCRIBER; (*one per terminal*)

define subread, subwrite;

var (*buffers, counters, and signals*);

procedure subread (var ch: char);
(*retrieve next character from input buffer when available*)
end subread;

procedure subwrite (ch: char);
(*deposit ch in output buffer and signal output*)
end subwrite;

process input;
(*input characters as long as input buffer is not full*)
end input;

process output;
(*output characters as long as output buffer is not empty*)
end output;

begin initialize variables; input; output

end SUBSCRIBER;
Figure 9
Trunk Device Module

device module TRUNK; (*one per trunk line*)

define trunkread, trunkwrite, writecontrol, postcontrol, waitcontrol;

use block;

var (*buffers, counters, and signals*);

(*procedures: 
  trunkread - get next block from input buffer
  trunkwrite - deposit block in output buffer
  writecontrol - deposit control character in output buffer
  postcontrol - save control character
  waitcontrol - retrieve posted control character*)

process input;
  (*input next character(s) into buffer and determine type of character*)
end input;

process output;
  (*output next control character or data block*)
end output;

begin initialize variables; input; output

end TRUNK;
passes a message identifier and estimated size as parameters. Createf allocates (or at least commits) adequate space, waiting if necessary until space is available. A directory entry is then initialized and a filename is assigned and returned to the calling process. The IN-CONTROL process subsequently fills the file by calling writef, specifying the filename and data block (of type block declared earlier). The writef procedure stores the block in the next allocated external record by calling AUXMEM. A file is "closed" by calling endwritef; its purpose is to allow MEMORY to deallocate any space originally requested (via createf) but not actually used.

Message files are read by OUTCONTROL processes by calling readf, specifying the filename, block (record) number, and storage variable used for returning the block. MEMORY maps the filename and block number into a sector address and offset and calls AUXMEM to perform the read. The block number must be specified on calls to readf because many OUTCONTROL processes may simultaneously be reading the same file (messages may have multiple destinations). Once all file reading is completed, SWITCH calls destroyf, which frees the space occupied by the file and, if necessary and possible, awakens a process waiting within createf.

AUXMEM is a device module within MEMORY that schedules and performs read and write operations on auxiliary memory. AUXMEM defines one procedure, perform IO, that takes an operation, sector, offset, and buffer and performs the operation (read or write). PerformIO requests a turn on the device; synchronizes with the device driver, then releases its turn. Device scheduling is handled by internal procedures that implement a scheduling discipline appropriate for the actual type of
Figure 10
Memory Interface Module

interface module MEMORY;
define createf, writef, endwritef, readf, destroyf;
use block;
(*variables - directory, free space, synchronization*)
(*utility procedures - spaceavail, request, release- for managing free space*)

procedure createf (msgid, size : integer; var filename : integer)
(*create file to store up to size blocks of msgid;
return name in filename*)

procedure writef (filename : integer; buffer : block);
(*store buffer as next record in filename*)

procedure endwritef (filename : integer);
(*release space allocated to but not used by filename*)

procedure readf (filename, blocknumber : integer; var buffer : block)
(*fetch the record given by blocknumber from filename and store it in buffer*)

procedure destroy (filename : integer);
(*destroy filename and release its space*)

device module AUXMEM;
define performIO; use block;
(*variables - sectorbuffer(s), synchronization*)
(*scheduling procedures - requestturn, releaseturn*)

procedure performIO (operation, sector, offset : integer;
var buffer : block);
(*schedule IO and then signal IO*)

process driver;
(*wait for IO; perform operation; signal done*)
begin (*initialize*)
end AUXMEM:
begin (*initialize free space etc.*)
end MEMORY;
device used. Notice that many processes could be waiting within AUXMEM. Only one at a time can be waiting for the driver to complete, though. While all are waiting and the device is servicing a request, future requests for file access can be handled by MEMORY and queued for scheduling within AUXMEM. The exclusion and synchronization mechanisms of Modula's interface and device modules make this possible.

The other interesting interface module is LINEOUTPUT. It schedules and controls output activity via three operations: insertoutput, receiveoutput, and doneoutput. For each output line (trunk or subscriber), LINEOUTPUT maintains a list of output messages where each list element identifies a message by its HEADERS index and priority. Each output list is ordered by priority. The insertoutput operation is called by SWITCH, once for each output destination; it inserts the new message in the appropriate place in the output list. If the new message is of higher priority than the one previously at the head of the list, a locally declared but exported (via define) pre-empt flag is set. This will cause the appropriate OUTCONTROL process to stop working on what it is doing and call receiveoutput to fetch the new, higher priority output message. Each OUTCONTROL process is therefore always working on the message at the head of its output list.

Receiveoutput is called by each OUTCONTROL process either when it is ready to output a new message or when it has found the pre-empt flag set. If no messages are available, the OUTCONTROL process waits for one to be inserted. When one is available, the header is retrieved from HEADERS and returned to the OUTCONTROL process. Receiveoutput leaves the output message on the output list so that it can be received again if it gets pre-empted. Once output is complete, an OUTCONTROL
process calls doneoutput, which deletes the appropriate entry from the process' output list.

The other four interface modules connect INCONTROL and OUTCONTROL processes to the SWITCH process. LINEINPUT is a simple message passing module that has two operations, sendheader and receiveheader, used to pass headers from INCONTROL processes to SWITCH. NOTICE implements a bounded buffer of notices for SWITCH. It has two operations, post and receivenotice, that synchronize with each other in the usual, bounded-buffer fashion (see [3] for example). REPLY is similar to NOTICE and has two operations, give and receivereply. All headers of active messages are stored in HEADERS by calling the enterheader operation. The header can be subsequently retrieved, updated, and deleted by calling the other three HEADERS operations.

The operations and parameters for each interface module are summarized in Figure 11. The comment with each operation summarizes its role and actions.

5.4 Processes

In this section, outlines of the three types of message switching system processes are given. INCONTROL processes read from device modules and build input messages. Because there are two types of devices in our prototype system, there are two types of INCONTROL processes. The outline for a subscriber input controller is shown in Figure 12. It reads characters one at a time and then takes certain actions depending on where it is in processing an input message. The three cases in Figure 12 illustrate the three main actions. Exceptions, such as an error or cancellation, are not illustrated, however.
Figure 11
Summary of Interface Module Operations

LINEINPUT -

procedure sendhead (hd : header);
(*save header*)

procedure receivehead (var hd : header);
(*retrieve next header*)

NOTICE -

procedure post (kind, data : integer);
(*save kind and data; signal receivenotice*)

procedure receivenotice (var kind, data : integer);
(*fetch next notice when available*)

REPLY -

procedure give (linenumber, data : integer);
(*save data for INCONTROL process of linenumber*)

procedure receivereply (linenumber : integer; var data : integer);
(*retrieve data saved for linenumber when available*)

HEADERS -

procedure enterheader (hd : header; var index : integer);
(*save header; return location in index*)

procedure retrieveheader (var hd : header; index : integer);
(*return copy of header at location index*)

procedure updateheader (hd : header; index : integer);
(*store hd at location index*)

procedure deleteheader (index : integer);
(*delete header at location index*)

LINEOUTPUT -

procedure insertoutput (line, msgindex, priority : integer);
(*store (msgindex,priority) at specified priority on
 output list of line; set pre-empt flag if necessary*)

procedure receiveoutput (line : integer; var hd : header);
(*return header of first message on output list of line
 as soon as one is available*)

procedure done (line : integer)
(*delete first entry from output list of line *)
Figure 12
Subscriber Input Control Process

process SUB_INCONTROL; (*one per SUBSCRIBER device module*)
(*variables for block, header, character, status, etc.*)

begin status := "in head";

loop subread(character); (*get next input char*)

case status of

"findstart" : begin
(*look for start of message sequence*)
if start of message then status := "inhead"
end;

"inhead" : begin
(*store character in appropriate header spot*)
if end of header then
sendhead(header); post ("start of input")
receivereply(); (*from REPLY*)
createf(); (*MEMORY file*)
status := "in body"
end
end;

"in body" : begin
(*store character in block*)
if end of input message then
writef(); (*store last block*)
endwritef(); (*in MEMORY*)
post("end of input"); (*NOTICE*)
elseif end of block then writef( )
end
end

end (*of case*)

end (*of loop*)

end SUB_INCONTROL
Trunk input controllers are similar to subscriber input controllers. They use the interface modules of Section 5.3 in the same way and take basically the same actions. The differences are that trunks transmit blocks instead of characters and require numerous control characters. The main point of interest is that the difference between the trunk and subscriber controllers results from and only affects the associated device module. The remainder of the switching system (MEMORY, SWITCH, etc.) is unaware of the difference.

OUTCONTROL processes transmit output messages from MEMORY to device modules. Again there are two types of output controllers, one for each type of output line. An outline of a subscriber output control process is shown in Figure 13. A trunk output controller is similar, differing only in the ways described above for the input controllers. Each SUB_OUTCONTROL process executes a loop, once for each output message whose header is received from LINEOUTPUT. It first formats and writes an output header. It then transmits the body of the message a block at a time. Each block is read from MEMORY and then written to the appropriate device module. On completion of output, LINEOUTPUT's done operation is called and the output process loops to receive the next message when it is available.

The most interesting aspect of output controllers is the way they handle pre-emption. Because a new output message of higher priority may become available while SUB_OUTCONTROL is busy with another message, it must detect the need for pre-emption and be able to stop what it is doing. Modula provides for an efficient solution to both problems. Each output controller periodically checks the appropriate pre-erpt flag exported from LINEOUTPUT and, if set, exits the inner
process SUB_OUTCONTROL; (*one per SUBSCRIBER device module*)
(*variables for header, block, and counters*)
begin (*initialize variables*)
loop
  loop (*inner loop is executed once for each output message
       outer loop allows escape from inner loop when pre-
       emption occurs*)
  receiveoutput (line, hd); (*from LINEOUTPUT*)
  (*format output message header*)
  repeat (*for each header character*)
      subwrite (character)
  until end of header;
  repeat (*for each data block of output message*)
     when pre-empt [line] do exit;
     (*exit inner loop on pre-emption*)
     readf( ); (*from MEMORY*)
     repeat (*for each block character*)
      subwrite(character);
     until no more characters;
     until no more blocks;
  done (line); (*to LINEOUTPUT*) post ("done")
end (*of inner loop*)
end (*of outer loop*)
end SUB_OUTCONTROL
loop statement via the Modula when statement. As shown in Figure 13, the flag is checked just before each read from MEMORY, hence once for each block of data. This could readily be changed to character level checking merely by moving the when statement inside the innermost (third) repeat statement.

The final system component is the SWITCH process which directs all activity. It is outlined in Figure 14. All communication to SWITCH is via the NOTICE interface module. For each notice, SWITCH executes a case statement as shown. A "start of input" notice signals the presence of a new input header which is received from LINEINPUT and stored in HEADERS. An "end of input" notice causes SWITCH to generate an output command for each destination named in the message's header. A "done" notice signals output completion for one destination; once all destinations have sent "done" notices, the message header and file are destroyed. In an actual system, other notices will also be handled by SWITCH, for example to stop a line, cancel a message, or signal an error.

6.0 Summary and Evaluation

The design of the message switching communication system has been described here in the order in which it was developed. We feel that this three stage design process (external specification, function specification, and program implementation) represents a valuable, structured approach to the design of large software systems. We also feel that Modula is an excellent design language for multiprogramming systems. One of the easiest ways to tell if a language is suited to
Figure 14
SWITCH Process

process SWITCH;
  (*variables for names of active messages (HEADERS indices),
   output directory, header, kind and data from NOTICE, etc.*)
begin (*initialize variables*)
loop receivenotice(kind, data);
  case kind of
    "start of input" : begin
      receivehead(head); (*from LINEINPUT*)
      enterheader(head,index); (*to HEADERS*)
      (*store index in active names, etc.*)
      give( ) (*REPLY*)
      end;
    "end of input" : begin
      retrieveheader(head, index);
      repeat (*for each destination*)
        insertoutput( ); (*in LINEOUTPUT*)
        until no more destinations
      end;
    "done" : begin
      decrease output count;
      if output complete then
        delete header( );
        destroyf
      end
      (*other cases for exceptions, cancel, etc.*)
    end (*of case*)
  end (*of loop*)
end SWITCH
the problem is to see if it helps or hinders the refinement of system functions into a program implementation. A good target language should guide the implementation by providing a structured framework within which to work. Modula proved to be ideal for this problem; we suspect it will be ideal for numerous other problems as well. The building blocks of Modula—processes and modules—were both appropriate and easy to use. Interface modules made it easy to implement and reason about process interaction, while device modules provided a natural, encapsulated means for implementing device interfaces.

To be more specific, Modula is a good multiprogramming language in four major respects. First, processes and interface modules (or monitors) are now recognized as basic concepts of parallel systems. In a system having parallel activity, it is important that each concurrent computation be clearly delineated. The process construct serves this purpose. In addition, system processes need to share information, often with exclusive access, and need to synchronize with each other. Interface modules serve these purposes.

The second positive aspect of Modula is the utility of device modules as a means for representing devices. It is the device module concept more than any other that, in our opinion, makes Modula the most attractive multiprogramming language available today. A device has much in common with any abstract data type: it accesses storage, defines operations, and requires exclusion and synchronization. Device modules, as a special kind of interface module, provide all these facilities. They give user processes a procedural interface to devices, implicitly provide exclusion, have signal variables for synchronization, and have driver processes to achieve
IO parallelism. They also allow details of device handling, such as buffer management, to be hidden from the user. In fact, Modula makes it possible to cleanly separate the file system functions in MEMORY from the device functions in AUXMEM. Finally, the device module construct makes it possible to control two quite different types of IO devices, subscribers and trunks, while providing a very similar interface to the controller processes.

The third major benefit of Modula is the ease and efficiency with which output pre-emption is implemented. The pre-emption problem occurs in many parallel systems so a multiprogramming language must make it easy to handle. The specific Modula features of value here are the ability to export read-only variables from modules and the ability to exit loops via the when statement. The OUTCONTROL processes test for pre-emption merely by checking a flag; if this flag were not accessible outside of LINEOUTPUT, output controllers would have to call a LINEOUTPUT procedure each time they wanted to check a flag. A benefit of the when statement (or any similar loop exit facility) is that changing the grain of pre-emption, namely the frequency at which the flag is checked, merely involves moving the when statement in output controllers. Finally, note that pre-empted messages could be easily restarted by keeping them on the output list in LINEOUTPUT.

The final positive aspect of Modula is its apparent execution efficiency. The storage and time required by the Modula kernel is minimal [10]. Consequently most of the space and time used by the message switching or any other system should result from functions of the system itself.

Despite its power, Modula is deficient in four respects as a
general language for the design and implementation of parallel systems. It is intended for small systems that reside entirely in main memory. As such, it does not provide tools for implementing large systems requiring virtual memory, dynamic creation and destruction of processes and modules, executable files, or dynamic access control. Modula serves its intended purpose well but further research is needed before we have more powerful multiprogramming languages suitable for large systems such as contemporary time-sharing systems.

A second deficiency is that two features of Modula exported variables and driver processes, create problems for verification because both lead to non-exclusive access to shared variables. Read-only variables, as noted, make it possible to efficiently handle pre-emption. Driver processes in Modula have higher priority than regular processes and maintain control after executing a send operation on a signal variable so that they execute whenever possible. Both features increase execution efficiency but at the price of increased verification complexity. They are specific examples of a general, unresolved dilemma of programming language design. To verify that the use of a read-only variable is correct, it is necessary to show that assertions in "reader" processes do not interfere with those in the interface module that can change the variable. To verify the correctness of a device module, it will be necessary to show that driver processes do not interfere with procedures in the same device module. Unfortunately, as becomes evident from a close reading of [6], it is not easy to prove that two processes are interference-free.

The third deficiency is that Modula does not allow processes or modules to appear in type declarations. As a result, one device
module, input controller, and output controller must be declared for each subscriber line even though the actions of each are identical. The same problem occurs with trunks. As long as code is shared, this does not affect efficiency because distinct names and storage space are obviously required. The readability and clarity of a full listing of the system would be much worse, however. Concurrent Pascal, by using a different access control and activation scheme, does not suffer this defect [2].

The final deficiency of Modula as a general multiprogramming language is its method for specifying scope rules. Define and use allow each module to export and import what it wants. Since a module is used to provide a service, it is natural for it to define which objects are accessible to others. For access security, however, a module should be told what it can use rather than take what it needs; it is more appropriate for the declarer of an object to specify which other blocks can access it. The general problem of access control in parallel programs as well as two specific language proposals are described in [5].

The above four deficiencies must be removed in a language intended for general systems programming. They should not, however, obscure the fact that Modula is the most powerful multiprogramming language currently available and that it meets its stated goals. The message switching system described here is but one example of its utility.

Acknowledgement

Sera Amoroso and Derek Morris proposed this application and served as sounding boards for the design as it was developed. Alan Demers and David Gries carefully read an earlier version of this paper and provided many constructive suggestions.
Bibliography


