AD-A105 200 KANSAS STATE UNIV MANHATTAN DEPT OF COMPUTER SCIENCE F/G 9/2
THE STRUCTURE OF THE NADEX OPERATING SYSTEM.(U)
DEC 79 R YOUNG, V WALLENTINE
UNCLASSIFIED TR-79-12 NL

1 of 1
AD
A105200

END
DATE
FILMED
10-81
DTIC

LEVEL II

12

AD A105200

(6)

THE STRUCTURE OF THE NADEX OPERATING SYSTEM

(10) Robert Young
~~and~~
Virgil Wallentine

DTIC
S ELECTE D
OCT 0 6 1981
E

(12) 53

TR-79-12

(14) TR-79-12

Computer Science Department
Kansas State University

(11) 1 Dec~~ember 1~~979

*This research was supported in part by grant number P-16160-A-EL from the Army Research Office and the Army Institute for Research in Management Information and Computer Science.

DTIC FILE COPY

81 10 2 122

391123

Table of Contents

1.0 Introductory NADEX Concepts

NADEX is an acronym for Network ADoptable Executive. NADEX supports the building of software configurations which consist of a general graph of communicating nodes. These nodes may be sequential or concurrent programs which access NADEX services through a native PREFIX. The PREFIX concept was originally defined by Per Brinch Hansen as an interface to the SOLO ~~[1]~~ operating system. The NADEX Native PREFIX is the interface to the NADEX Core OS and provides data flow abstractions to the program running in a node. These operations permit each program (running in a node) to exchange messages with other nodes in a software configuration via full-duplex data transfer streams.

In this document, we first present the concept of a software configuration. We then present the general structure of NADEX. Finally, we describe the function of each module of the NADEX Core OS as it is written in Concurrent Pascal [1].

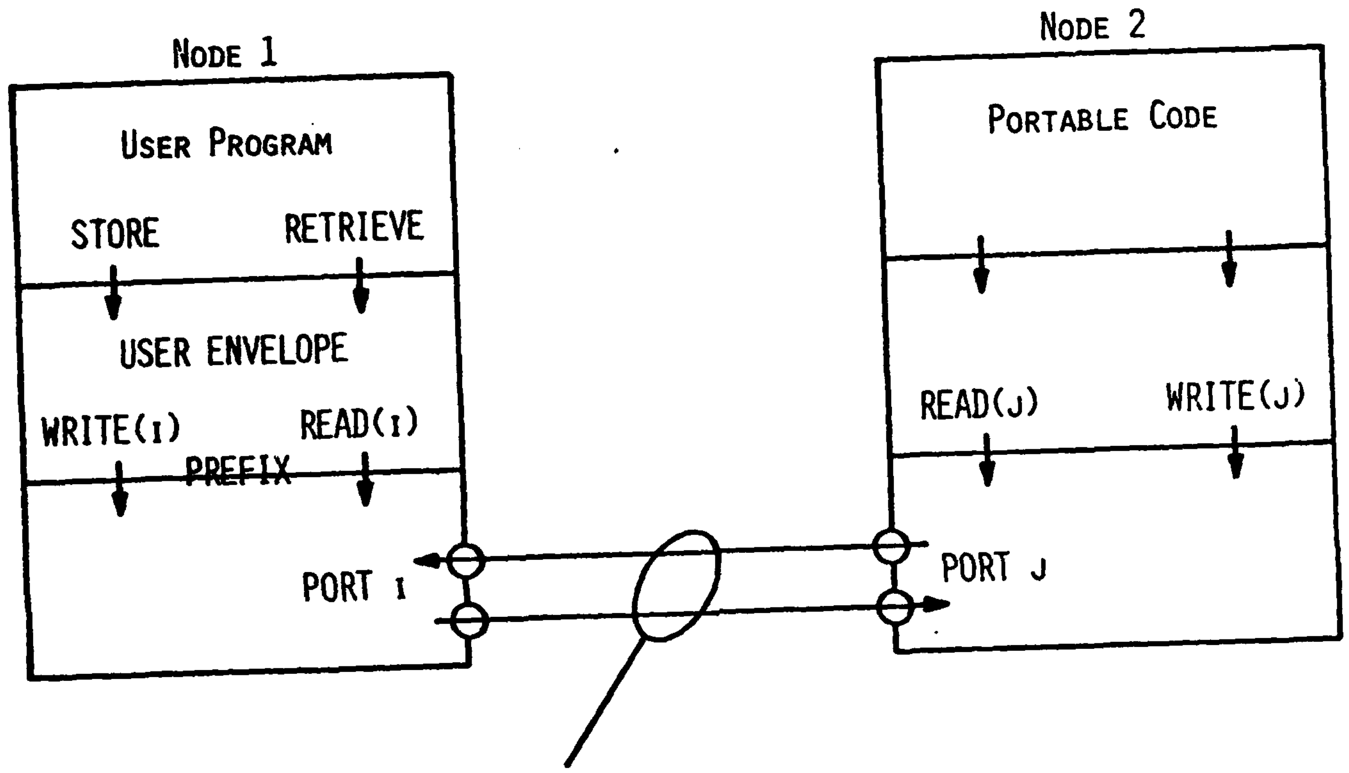
2.0 Software Configurations

2.1 Configuration Properties

A configuration consists of a collection of nodes connected by data transfer streams (DTS's). Nodes can be user programs (both sequential and concurrent languages such as SPASCAL and CPASCAL), file access nodes (for accessing files within the NADEX file system), I/O device access nodes (for accessing I/O devices not supported by the NADEX file system), or external configurations such as subsystems.

Nodes within the configuration are connected by DTS's which are also called connections. Each connection consists of two bi-directional components--data and parameter. The data component transfers data in page-sized blocks (a page is 512 bytes) and interfaces to the user program at the page, logical record, or character level. The parameter component transfers small parameter blocks typically used for control information. The data and parameter components are totally independent. The two directions of each component are independent in the sense that each direction has its own queue, but the user protocol restrictions are defined in terms of the bi-directional components.

For the purposes of these discussions, we will speak of a node issuing reads and writes to a port which is local to the node. The connection of these ports forms the Data Transfer Stream. This is illustrated in Figure 1. Table 2.1 contains the PREFIX which implements these operations. These should be assumed to be read-page and write-page



1. Connections (I,J) established by configuration command processor
2. Connections can be implemented in common memory, DMA connection or loose coupling

Figure 1

User Program Communications in Software Configurations

Table 2.1

```
****************************************
*                                      *
*          NADEX NATIVE PREFIX         *
*                                      *
****************************************

CONST PAGE_SIZE = 512;          "SIZE OF DATA PAGE"
      PARM_SIZE = 32;           "SIZE OF PARAMETER BLOCKS"
      MAX_DTS = 40;             "MAX GLOBAL DTS ID"
      MAX_PORT = 10;            "MAX PORT ID"
      SVC1_BLOCK_SIZE = 24;     "SIZE OF SVC 1 PARM BLOCK"
      SVC7_BLOCK_SIZE = 26;     "SIZE OF SVC 7 PARM BLOCK"

TYPE PAGE = ARRAY [1..PAGE_SIZE] OF BYTE;
     PARAMETER = ARRAY [1..PARM_SIZE] OF BYTE;
     SVC1_BLOCK = ARRAY [1..SVC1_BLOCK_SIZE] OF BYTE;
     SVC7_BLOCK = ARRAY [1..SVC7_BLOCK_SIZE] OF BYTE;

TYPE DTS_INDX = 1..MAX_DTS;    DTS_INDX0 = 0..MAX_DTS;
     PORT_INDX = 1..MAX_PORT;  PORT_INDX0 = 0..MAX_PORT;

TYPE DTS_SET = SET OF DTS_INDX;

TYPE BUF_TYPES = (PARM_BUF, DATA_BUF, NIL_BUF);
                           "BUFFER TYPES"

TYPE REQ_CODES = (REQ_OK "0", REQ_NODE_ABORT "1",
       REQ_DTS_ABORT "2", REQ_DEFER "3", REQ_UNRES_DTS "4",
       REQ_PROT_ERROR "5", REQ_BAD_PORT "6");
         "PREFIX DTS OPERATION RETURN CODES"

PROCEDURE READ_CHAR (PORT:  PORT_INDX; VAR C:CHAR);

PROCEDURE WRITE_CHAR (PORT:  PORT_INDX; C:CHAR);

PROCEDURE READ_DATA (PORT:  PORT_INDX; VAR DATA:  UNIV PAGE;
                     VAR LENGTH:  INTEGER;
                     VAR RESULT:  REQ_CODES);

PROCEDURE WRITE_DATA (PORT:  PORT_INDX; DATA:  UNIV PAGE;
                      LENGTH:  INTEGER; CONDITIONAL:
BOOLEAN;
                      VAR RESULT:  REQ_CODES);

PROCEDURE READ_PARM (PORT:  PORT_INDX;
                     VAR PARM:  UNIV PARAMETER;
                     VAR RESULT:  REQ_CODES);

PROCEDURE WRITE_PARM (PORT:  PORT_INDX; PARM:  UNIV
PARAMETER;
                      CONDITIONAL:  BOOLEAN;
                      VAR RESULT:  REQ_CODES);

PROCEDURE MAP_PORT (PORT:  PORT_INDX; BUF_TYPE:  BUF_TYPES;
                        VAR RDTS:  DTS_INDX0;
                        VAR WDTS:  DTS_INDX0);

PROCEDURE AWAIT_EVENTS (VAR READ_WAITS, WRITE_WAITS:
DTS_SET;
                        VAR READ_READY, WRITE_READY:
DTS_SET;
                        VAR RESULT:  REQ_CODES);

PROCEDURE DISCONNECT (PORT:  PORT_INDX;
                      VAR RESULT:  REQ_CODES);

PROCEDURE FETCH_LUSER_ATTRIBUTES;

PROCEDURE SUBMIG_CONFIG;

PROGRAM FSS;
```

requests for the data component, and read-parm and write-parm requests for the parameter component. The blocking of character and logical record data into pages is handled by the prefix of the nodes and will not be discussed here. Unless otherwise specified all discussions apply equally to data and parameters, and no distinction will be made.

There are no structural restrictions on the graph formed by the nodes and connections (DTS's). In particular, it need not be linear (like SOLO [1] and UNIX [11]) or hierarchical. It need not even be acyclic as in AMPS [16] or connected. Nodes are not precluded from having connections to themselves. Thus, the configuration is described by a (labeled) undirected graph. The labeling occurs where each connection enters the two (not necessarily distinct) nodes it connects.

The user programs (as well as the various system routines which implement the other nodes) address the connections emanating from each node by DTS ids local to the node. These local DTS ids are also called port numbers. The meaning of the data stream associated with each port is defined by the program. Port numbers are generally assigned by the programmer starting with one (since the system will place a configuration-dependent upper limit on the port numbers for economy in table storage). These port numbers are the labels on the configuration graph.

The structure of a configuration is defined by a language which builds a file called a Configuration Descriptor File (CDF). The CDF defines the structure of the configuration and the type (user program, file access, etc.) of each node. When the user requests that a configuration be run (either through a terminal command or a command in a batch job), the CDF is used along with information from the command to construct a configuration descriptor. The configuration descriptor includes all of the information about the configuration including, for example, the names of the files to be accessed by the file access nodes. The configuration descriptor contains enough information for the system to allocate resources and run the configuration.

A typical language for building CDF's might include statements like 'DEFINE NODE 1 AS USER PROGRAM (filename),' 'DEFINE NODE 2 AS FILE ACCESS (filename),' 'CONNECT NODE 1 PORT 1 TO NODE 2 PORT 1.' More complete examples of configuration description languages are given in reference [15]. Thus, there are statements which define each node and the function it is to perform as well as those which define each connection. The node definitions may completely specify the function, or some information (such as filenames) may be left to be filled in from the command. The connection definition may include buffer allocation parameters (to be discussed later). The program which converts the CDF into a configuration descriptor is called the Command Processor and runs as a separate configuration.

2.2 DTS Implementation Considerations

The system places no interpretations on any of the data contained in the data and parameter buffers, except when they are used to communicate with a system node (such as file access or I/O device access). Thus, the user is free to design his own protocols. The system guarantees that data (and parameters) are delivered in the same order as they were written. Note that this applies to each component in each direction independently.

However, the user protocols implicitly define buffer allocation parameters which must be available to the system to ensure that deadlock due to buffer allocation can be avoided. For each bi-directional component (data and parameter) of a connection, there is a buffer allocation quantity called min. This designates the minimum number of buffers which must be reserved for this bi-directional component of the connection. The user protocol is related to this quantity in that it must operate such that it never requires more than min data items (data pages or parameters) in-transit at any one time. For example, if two connected nodes both issue write datas followed by read datas to the same connection, then min must be at least two. Normally, synchronized protocols only require a min of one. Mins are handled separately for parameter and data. Generally, min will be of concern only when data items are not read in the same order in which they were written over a single connection, or when cycles exist within the configuration graph. Note that min is an inherent property of user protocols (although some min´s are data dependent and possibly unbounded).

When the CDF is defined, the min values for each connection must be defined to the system. These values must be at least as large as the min required by the protocol. The system will default a min of one since most naturally-occurring protocols require only a min of one. The system will ensure as much as possible that the user protocol does not violate min, and will terminate the configuration if it does. If the user protocol violates the min restrictions, it is possible for the deadlock avoidance in the buffer allocation policies to fail and thusly for the configuration to deadlock.

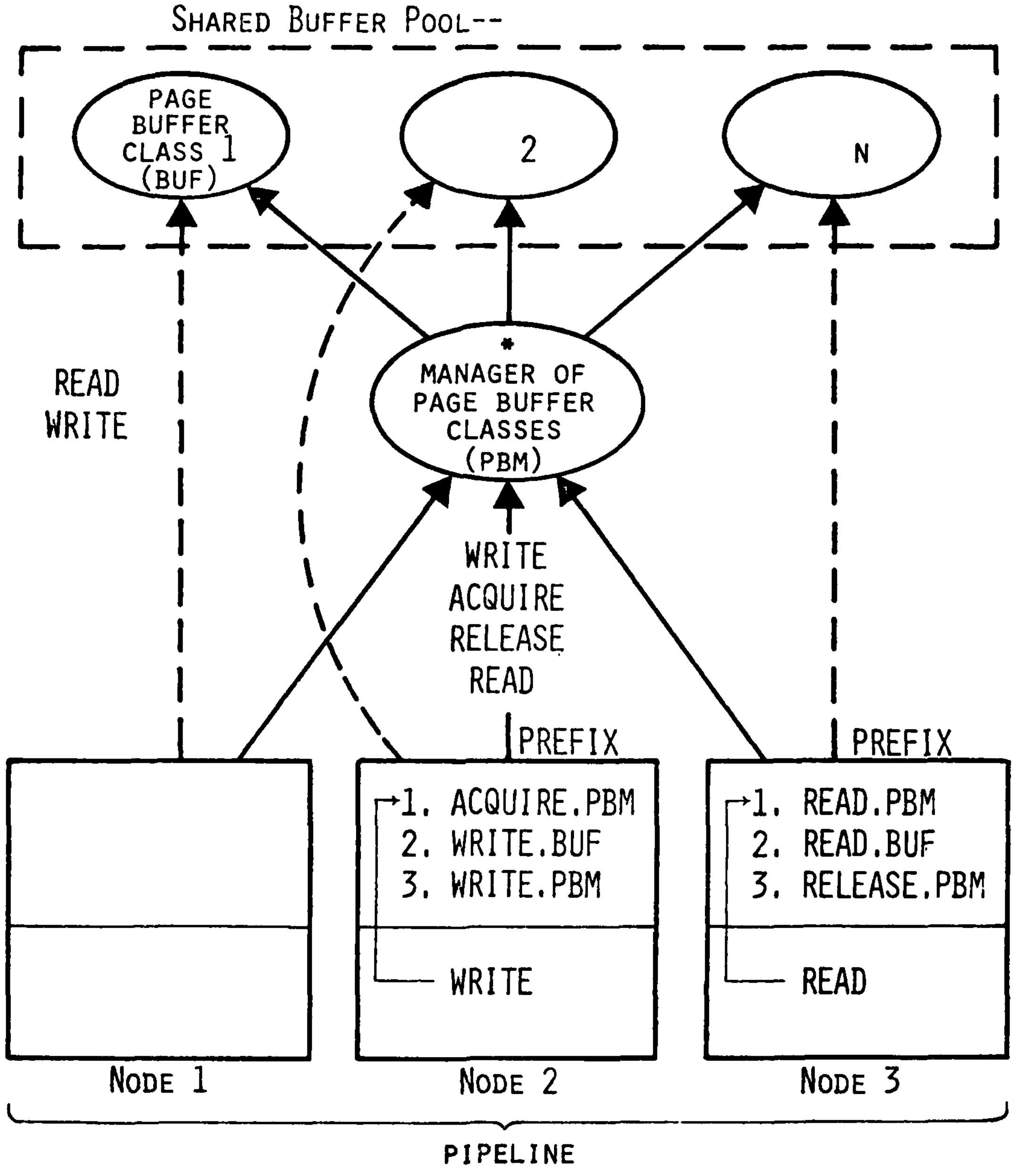
In addition, a max will be defined for each connection component. Max is an upper bound on the number of buffers which the system is constrained only in that it must be able to supply at least min buffers to each connection component. It also should not supply more than max buffers to each connection component. It may make its own decisions within those constraints as appropriate based on the relative data flow rates within the configuration. The default value for max will be the total number of buffers of the type allocated to the configuration. The number of data and parameter buffers to be allocated are also parameters in the CDF which default to the sum of the mins. Thus, the mins ensure that the user protocols can function without deadlock, and the max's allow the user to partition the buffer pool overriding the dynamic collocation policies of the system.

The DTSs associated with each configuration are implemented with a Pipeline Buffer Manager (PBM). Pipeline is a misnomer in this case but the PBM acronym has remained with the system historically. Functionally, the PBM is the Configuration Buffer Manager. It is implemented as a CPASCAL manager which manages parameter and data buffer classes. The PBM contains within it the queues of data items waiting to be read, pools of free buffers, buffer allocation information, and deadlock detection information.

The read and write functions are each implemented via a pair of calls to the PBM. A read is implemented by a read to the PBM which returns a reference to a buffer. The data is then extracted from the buffer and the buffer is released to the free pool by a release call. A write is implemented by a request which fills the free buffer, which is followed by a write which acquires a free buffer, which in turn is followed by a write which queues the buffer so that it can be read in sequence by the node at the other end of the connection. We illustrate this in Figure 2.

The permanent variables of each node contain a table which maps local DTS ids (port numbers) into PBM DTS ids. The PBM DTS ids are DTS ids which uniquely identify a connection within the PBM (and thus within the configuration for the configurations described thus far). The mapping

A) Mechanism of Buffer Allocation Within Pipeline



Legend:

*	GUARANTEES NO DEADLOCK ON BUFFERS SHARED IN PIPELINE
———	ACCESS RIGHT
- - -	ACCESS VIA REFERENCE

B) Conceptual Representation

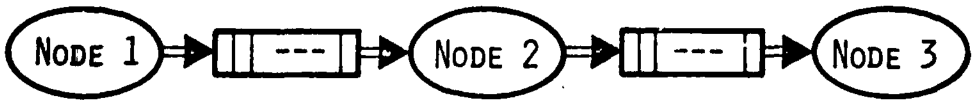


Figure 2: Pipeline Connections

table also contains an indicator that shows which end of the connection this port implements. (The ends of the connection are arbitrarily named by the system, and the names are used to identify which direction a particular data item is flowing). (As an implementation consideration, the PBM DTS ids are mapped into pairs of numbers, and the connection end determines whether the even or odd member of the pair is used.)

Thus far, we have described a basic self-contained undistributed configuration and the mechanisms (PBM and prefix) used to implement it. Note that each node is defined and programmed solely in terms of its ports and is not knowledgeable of the structure of the configuration. All interprocess communication is done via data transfer streams. These characteristics will prove quite useful later when the configuration is distributed.

2.3 Subsystems

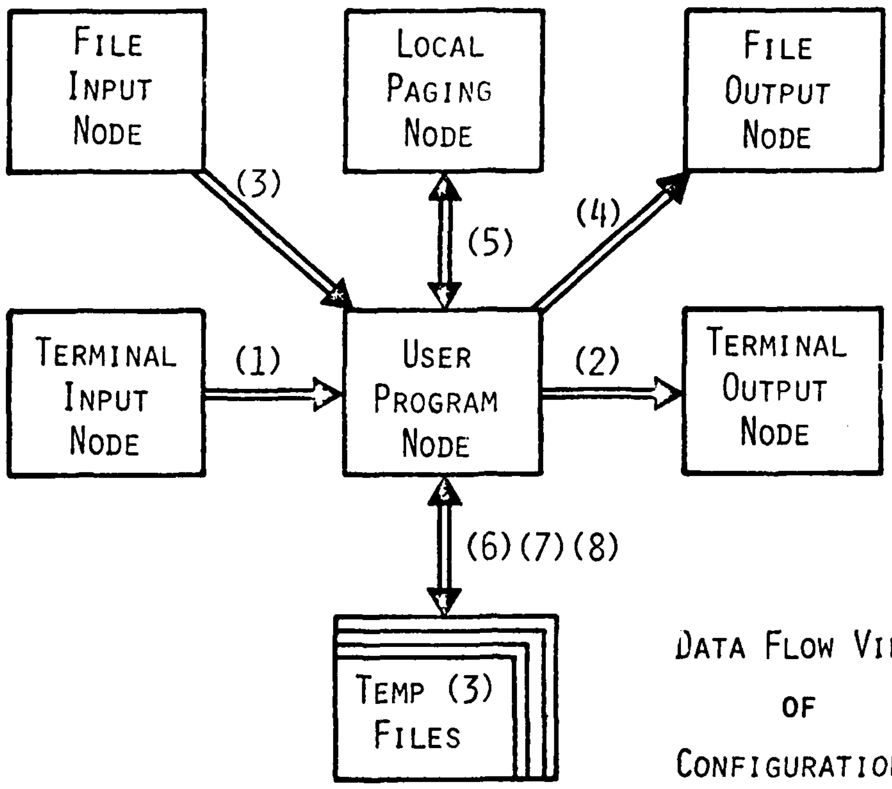
In addition to running isolated user configurations, NADEX allows user configurations to communicate with special configurations known as subsystems. As mentioned, subsystems are themselves configurations, but also have an interface to allow connection to nodes of user configurations. Typical uses of subsystems would be a data base management system, a file system, and an interconfiguration communications system.

A subsystem is unique in that it is not activated directly by a user command or a user batch job. Instead, it is activated whenever a configuration is started which requires the services of that subsystem. The subsystem then continues to run until there are no more active users (configurations). Then, depending on the subsystem, it may be automatically terminated or it may remain active waiting for additional users.

A subsystem serves multiple configurations concurrently and has much of the responsibility for multiplexing itself among its users. Furthermore, as user configurations are initiated and terminated, they can dynamically connect to and disconnect from the subsystem.

When a user describes a configuration, access to a subsystem is shown as a single node with connections from other nodes in the configuration. However, when the configuration is implemented, the system will not create such a node. Instead, the connections to the subsystem node in the user description will be connections between user nodes and the user interface of the subsystem. This is illustrated in Figure 3.

The user nodes communicate with the subsystem just as if it were another node. The normal DTS operations (read and write parameter and data) are used to implement a protocol defined by the particular subsystem. The user nodes themselves are not aware that they are communicating with a subsystem rather than with another node (which uses



Data Flow View of Configurations

Legend:

(1), (2), (3), (4)	sequential I/O streams
(5)	paged buffer protocol
(6), (7), (8)	random I/O protocol

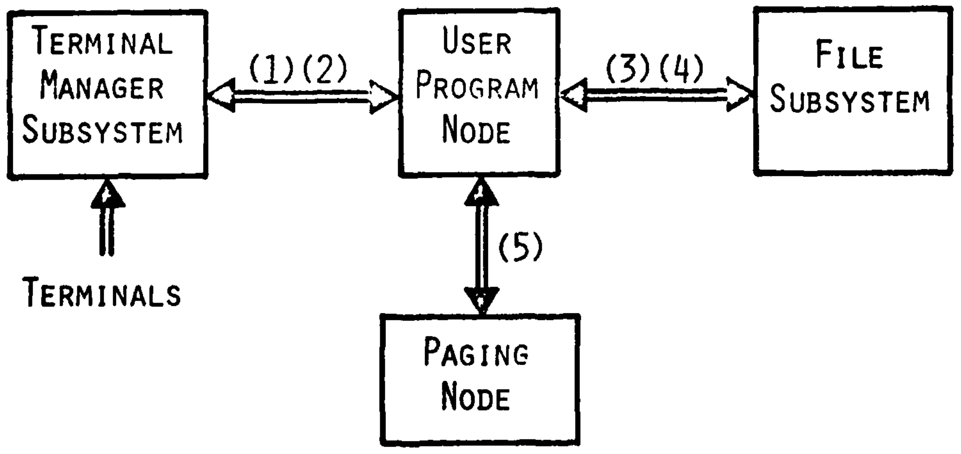


Figure 3
Implementation of Data Flow in Subsystem Connection

the same protocol, of course).

From the subsystem's point of view, there is a set of connections defined in the subsystem's configuration description called the user interface connections. These have one end which terminates within the subsystem (perhaps on a one-to-many basis) and the other end is initially left free. When user configuration is started, its connections to the subsystem will be implemented over some of these user interface connections.

The subsystem again uses the normal DTS operations to communicate with the various users which it serves. Since subsystems serve multiple users, they will typically be users of the multiple-condition wait, which waits for requests coming from the various users.

Note that neither the subsystem nor the user configuration is aware (at this point) that the subsystem is actually a subsystem. In fact, without changing any of the programming discussed so far, the collection of nodes which constitute a subsystem could be taken and placed in the user configuration. The connections from user nodes to the subsystem would be the user interface connections, and the two parts of the configuration would otherwise be independent. This allows a user to use a private copy of the subsystem within a user configuration if necessary. This is the recommended procedure for debugging subsystems. Note that no changes in any of the programming is required to move between these two modes.

2.4 Implementing Subsystems

In the implementation of a real subsystem, the subsystem exists just as any other configuration does. The only distinctions are the presence of the user interface connections and the lack of a user terminal or batch job which controls the configuration. In particular, the configuration does have a PBM which implements both its own internal connections and part of the connections to users over the user interface connections.

Once a user configuration has connected to a subsystem, we can speak of the user-subsystem connection as a single entity which was formed by matching a port in the user configuration description which terminated at the subsystem node with a user interface port in the subsystem. However, this connection is really implemented by two connections--one in the user PBM and one in the subsystem PBM. This complication is necessary to ensure that all of the events which cause conditions in an await-condition request to become true occur within the caller's PBM.

In general, each side of the connection issues reads within its own PBM but issues writes to the other PBM. All buffers are allocated from the user PBM. Thus, a user writes to a subsystem by acquiring a buffer from the user PBM, filling it with data, and writing it into the subsystem PBM. A user reads from a subsystem by reading a buffer from the user PBM, copying out the data, and then releasing it into the user PBM. A subsystem reads from a user by reading a buffer from the subsystem PBM and then releasing it into the user PBM. A subsystem writes to a user by acquiring a buffer from the user PBM, filling it with data, and then writing it into the user PBM.

Note that the data-available condition for the user occurs in the user PBM, and the data-available condition for the subsystem occurs in the subsystem PBM. However, since all buffers come from the user PBM, the buffer-available-for-write condition always occurs within the user PBM. It is anticipated that subsystems will always use conditional writes into user PBM's. (Actually, it is the acquire that is conditional.) If such a request is rejected due to a buffer unavailability (either due to max excession or a depletion of the PBM free buffer pool), then the connection is flagged. Note that this can occur only if at least min buffers are currently queued for the user. When the user issues a read from this connection, the flag is interrogated. If it is set, then a status is returned (to the prefix routine) which causes it to call the subsystem PBM to inform it that buffers are (maybe) now available for this connection. That will cause a wait for buffer-available within the subsystem PBM for that connection to be marked as complete, and the subsystem will eventually retry the conditional write.

The user-subsystem connection is actually implemented as two separate DTS's--one in the user PBM and one in the subsystem PBM. For this reason, the prefix entries for this connection must have the PBM references and the PBM DTS ids for both parts of the connection, and thusly will use the appropriate ones depending on the type of call.

The await-condition facility allows the subsystem to exercise flow control among its various users. Since the subsystem can remove connections from the waiting sets and use its own algorithms to determine which connections to service first, it can implement whatever controls are necessary. Since all user-subsystem communication uses buffers from the various user PBM's, there are no problems with buffer allocation within the subsystem PBM due to the varying number of users and their requirements. The conditional write and await-multiple-condition facilities prevent the subsystem from even being forced to wait in a particular user's PBM or to wait for a request from a particular user.

Note that many-to-many connections are restricted in that each end of a connection must reside totally within one configuration. Thus, an end of a connection cannot be split between a subsystem node and other nodes in the description of a user configuration. The user end of a user interface connection in a subsystem cannot be split, although the subsystem end can and probably will be split between various nodes of the subsystem.

Subsystems are not prevented from being users of other subsystems. However, the restriction that a configuration cannot be initiated until all of the subsystems it uses are running imposes a hierarchy on subsystems. Since the user interface connections are driven by requests from user configurations, it is not possible to connect the user interface connections of two subsystems together.

Note that the deadlock detection algorithms cannot detect protocol violations for users communicating through subsystems (even if the communication takes the form of just requesting the same resource). This is due to the fact that internode dependencies are known only within the internal data structures of the subsystem and cannot be deduced from the various PBM data structures.

2.5 DTS (Port) Operations to Support Multi-server Subsystems

The read and write functions described thus far have been unconditional serial operations; that is, they request a specific operation on a specific port and do not return until that operation has been performed. (They can also terminate as a result of configuration termination due to errors.) Therefore, programs which use them are deterministic (assuming the nodes to which they are connected are deterministic) and do not use undefined variables.

In order to write programs which process several data streams concurrently and which are adaptive to the data flow rates in the various data streams, it is necessary to have conditional operations and multiple-event waits. (See Table 2.1.) As shown in Figure 4, a subsystem may have to service multiple ports whose requests are asynchronous. This condition, as noted also by Sunshine in UNIX [11], must be solved by tagging messages with the sender's identifier or with unique ports. Since each port can be identified with a particular function, and since tagged messages require the user node to de-multiplex messages in the data stream, the NADEX Native PREFIX provides a facility (AWAIT_EVENTS) to identify when and which DTS(s) require attention.

Basically, the DTS-related events which can occur involve either the availability of data items (due to a write from the other end of the DTS) or of buffers (within the min/max constraints) so that a write can be performed. These events also describe the only conditions under which a node can be delayed while executing a DTS-related operation.

A multiple-event wait prefix routine is defined which allows a user to wait for the occurrence of any one of a set of events. The events involve the availability of data items or that of buffers for the data and parameter components of each of the DTS's connected to the node. These are represented by four sets--data-input, parameter-input, data-output, and parameter-output-of local DTS ids. A DTS id is a member of the set if that condition will cause the wait to complete. Note that these are "remembered" events and are really conditions rather than

Example Systems Which Require ASYNCHRONOUS COMMUNICATION

Teleconferencing (TEMS) - Electronic Mail System

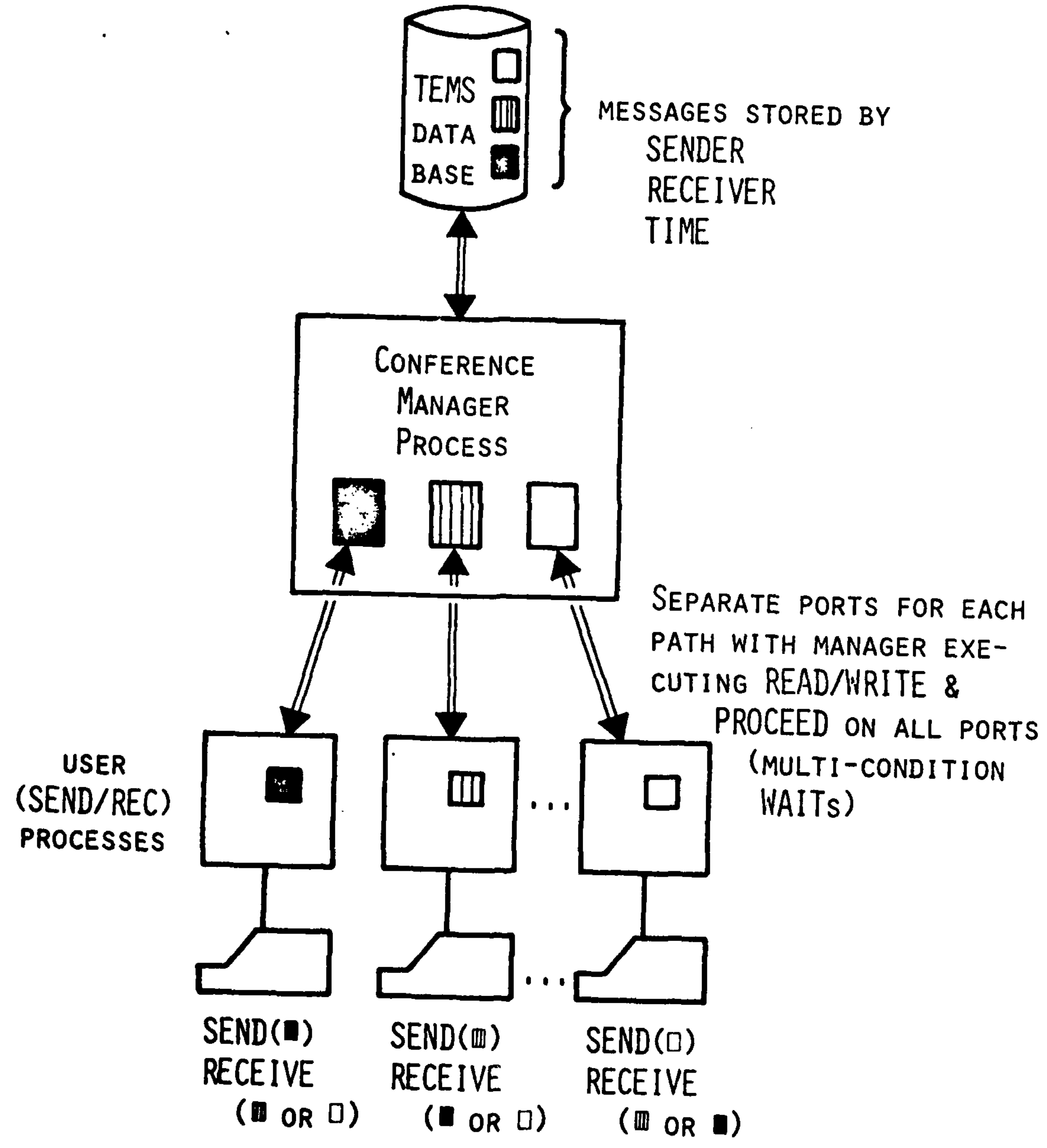

TEMS DATA BASE
MESSAGES STORED BY
SENDER
RECEIVER
TIME
CONFERENCE MANAGER PROCESS
SEPARATE PORTS FOR EACH PATH WITH MANAGER EXECUTING READ/WRITE & PROCEED ON ALL PORTS (MULTI-CONDITION WAITs)
USER (SEND/REC) PROCESSES
SEND(■) RECEIVE (▥ OR □)
SEND(▥) RECEIVE (■ OR □)
SEND(□) RECEIVE (▥ OR ■)

Figure 4

events. When the multiple-wait returns, the user will be provided with an indication all of the conditions which now hold. The user can then, based on those conditions, issue the appropriate unconditional request knowing that it will complete immediately.

The last statement was true for input requests but is not always true for output. Buffer availability is dependent on two factors--the connection will not exceed max, and there is a free buffer in the buffer pool. Clearly other nodes which share the same buffer pool can cause the second condition to become false after it has been signaled via a multiple-wait that it was true. The other node on a connection can cause the first condition to become false also. Therefore, it is necessary to have a conditional write which will return a status indicating whether or not a buffer could be allocated.

3.0 NADEX Structure

NADEX is an operating system in support of full-duplex data transfer streams (DTS's) between a general graph structure of nodes--software configurations. Each node can run concurrent or sequential programs. In this section we describe the structure of NADEX, its layering and the interfaces between the layers.

In Figure 5 we illustrate the structure of NADEX as a system of five layers. The outer ring is a possible set of subsystems in support of user configurations. The NADEX Core OS supports only DTS operations and configuration construction so that a system can be tailored to a user environment. The subsystems in the outer layer are such tailored functions. These subsystems are interconnected via the data transfer stream (data flow control) mechanism of layer 3. These DTS's are established by layer 4, and data is transmitted via the implementation of layer 3. Access to these services are provided to user programs using the NADEX PREFIX.

This NADEX Native PREFIX--interface 4 in Figure 5--is presented in Table 2.1. Its design objective was to present to the user a set of data flow abstractions. With these operations a user (or user envelope) can develop any protocol he wishes between nodes in a configuration. Examples of the use of this PREFIX are presented in reference [17].

In addition to DTS operations, a user or system program

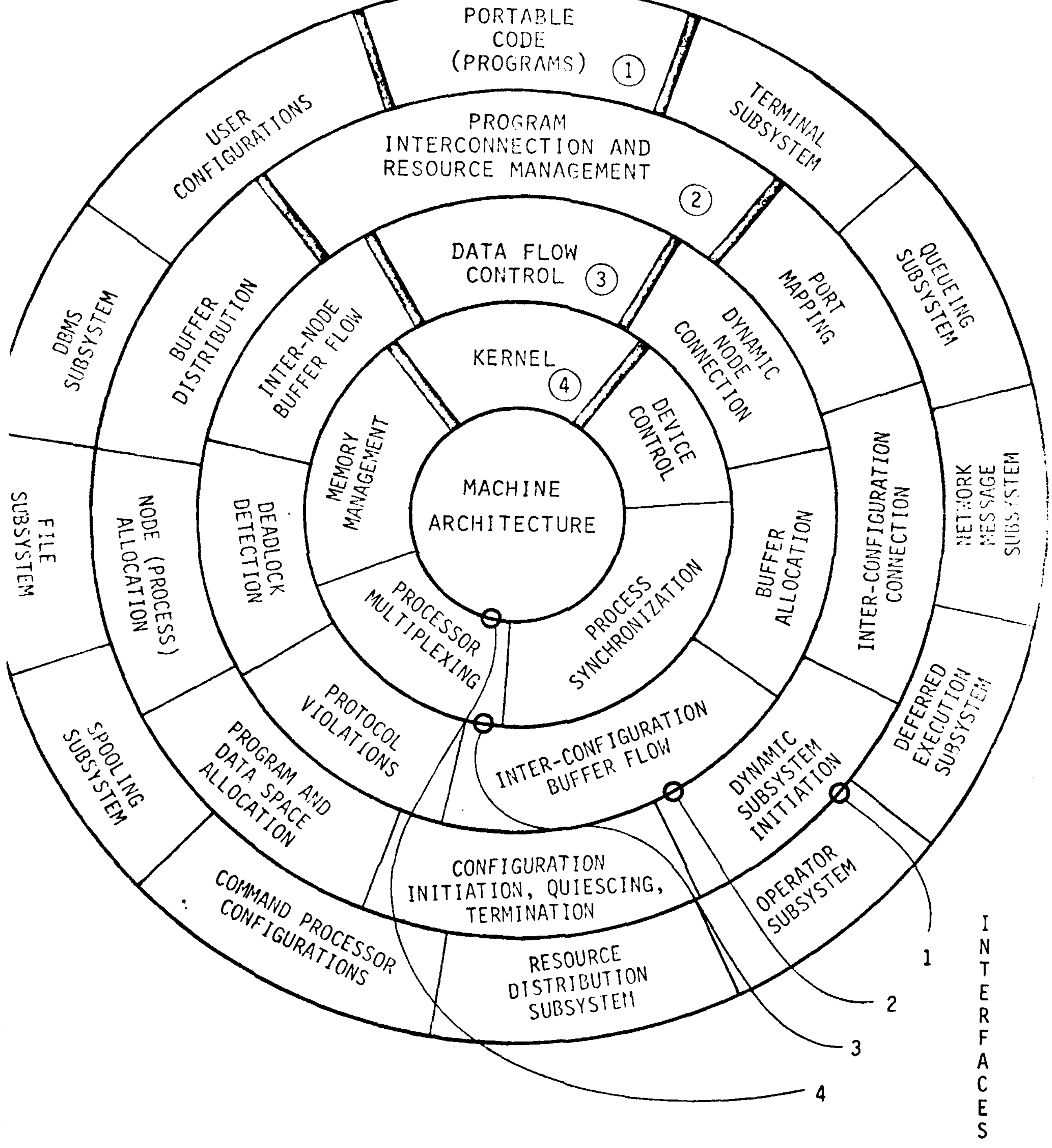

PORTABLE CODE (PROGRAMS) 1
TERMINAL SUBSYSTEM
QUEUEING SUBSYSTEM
NETWORK MESSAGE SUBSYSTEM
DEFERRED EXECUTION SUBSYSTEM
OPERATOR SUBSYSTEM
RESOURCE DISTRIBUTION SUBSYSTEM
COMMAND PROCESSOR CONFIGURATIONS
SPOOLING SUBSYSTEM
FILE SUBSYSTEM
DBMS SUBSYSTEM
USER CONFIGURATIONS
PROGRAM INTERCONNECTION AND RESOURCE MANAGEMENT 2
PORT MAPPING
INTER-CONFIGURATION CONNECTION
DYNAMIC SUBSYSTEM INITIATION
CONFIGURATION INITIATION, QUIESCING, TERMINATION
PROGRAM AND DATA SPACE ALLOCATION
NODE (PROCESS) ALLOCATION
BUFFER DISTRIBUTION
DATA FLOW CONTROL 3
DYNAMIC NODE CONNECTION
BUFFER ALLOCATION
INTER-CONFIGURATION BUFFER FLOW
PROTOCOL VIOLATIONS
DEADLOCK DETECTION
INTER-NODE BUFFER FLOW
KERNEL 4
DEVICE CONTROL
PROCESS SYNCHRONIZATION
PROCESSOR MULTIPLEXING
MEMORY MANAGEMENT
MACHINE ARCHITECTURE
1
2
3
4
INTERFACES

FIGURE 5
NADEX STRUCTURE

(command processor) can initiate another configuration via the NATIVE PREFIX. In order to support this spin-off of configurations, the NADEX CORE OS (layers 2 and 3) needs a representation of the configuration. This is represented by a Configuration Descriptor (CD). This CD is described in reference [17], and it is submitted to NADEX through the SUBMIT_CONFIG call.

The NADEX Core OS (layers 2 and 3) is implemented in CPASCAL/32. Therefore, its kernel (layer 4 in Figure 5) is the CPASCAL/32 kernel. The functions of this kernel and its interface (interface 3 in Figure 5) in support of NADEX is described in reference [18]. The remainder of this document consists of a discussion of the structure of layers 2 and 3 as they are implemented in Concurrent Pascal.

4.0 NADEX Operation

In this section, we discuss the structure of the NADEX Core OS module by module and then trace the operation of the NADEX system from system initialization through the initiation and termination of a user configuration. Within this document, various acronyms will be used for system component names which correspond to names used in the NADEX code. For more detailed information on the topics discussed here, refer to the NADEX code listing and associated support routines.

4.1 Concurrent Pascal (CPASCAL) Language Extensions

Several extensions were made to Concurrent Pascal to support the implementation of NADEX. These extensions were made under two conditions. First, any change must be absolutely necessary for efficiency of operation in both time and space. Second, the basic precepts of CPascal (compiler checking for time dependent errors) must not be violated.

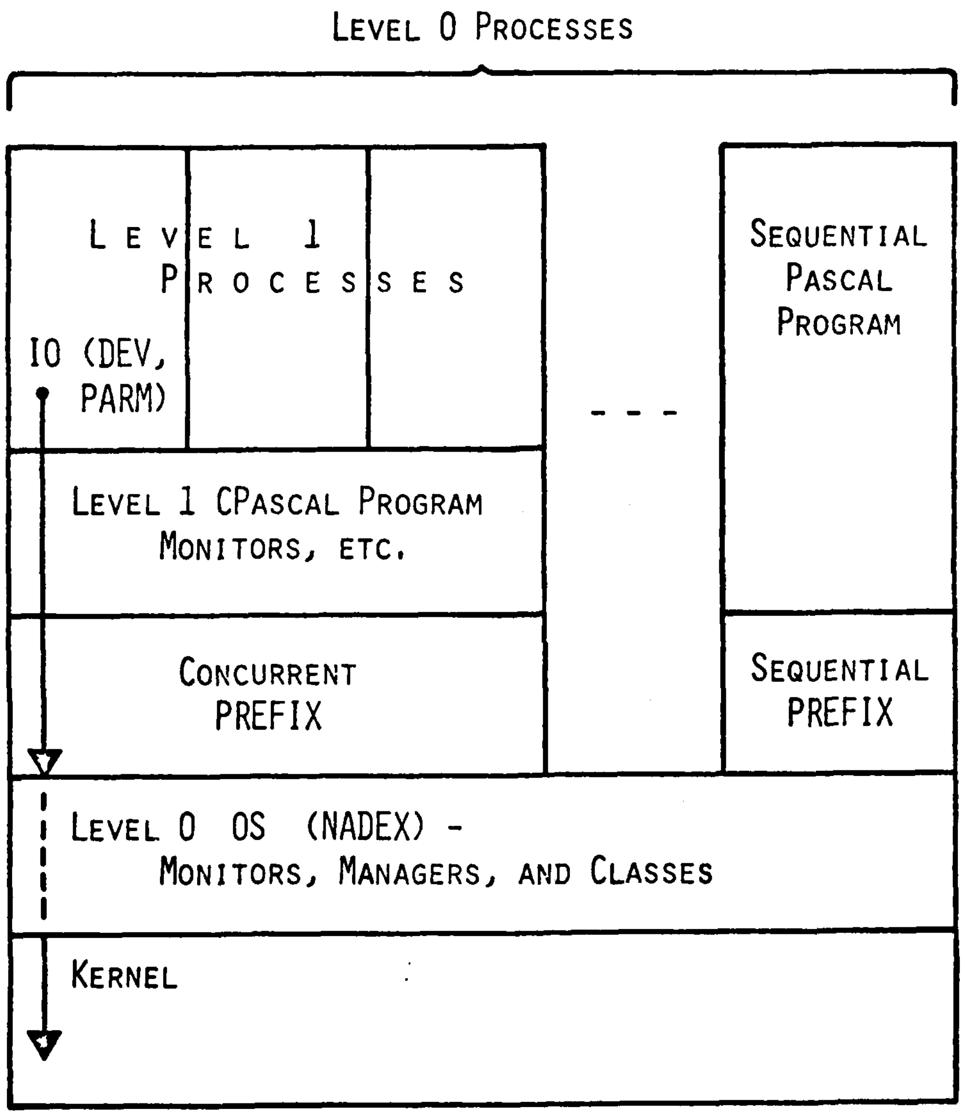
The first extension was to provide a mechanism for dynamic allocation of resources--buffers and memory. The manager concept of Silberschatz, et. al. [19] was chosen but with a syntax more in keeping with Sequential Pascal. Pointers to system components were introduced so that resources could be represented as system components (monitors and classes) and dynamically assigned to various processes as these resources were needed. (See Figure 2.) A "transfer-only" assignment of pointer values (via the assignment statement or vice parameter passage) to class types prevents the addition of any new time dependent errors in CPascal.

Second, we added records as system types. This was necessary to pass large data structures between processes without excessive copying. Pointers to records are treated as pointers to classes in terms of "transfer-only" assignment. Records are used in NADEX whenever an excessive number (and sometimes an undetermined number) of different encapsulations of the data are required. This would have forced each class to have an excessive number of entry procedures. In fact, there are times when it is impossible to anticipate what encapsulation is necessary.

Third, we extended the kernel to support hierarchical Concurrent Pascal programs. This permits concurrent programs to be executed under an operating system--a Concurrent Pascal Virtual Machine. This is documented in reference [16] and illustrated in Figure 6.

4.2 The NADEX Core OS as Concurrent Pascal Program

The NADEX system initial process initializes all of the NADEX system components, activates configuration processes for the two terminals used for testing, and then terminates. The basic functions of each system component along with the



LEVEL 0 PROCESSES
LEVEL 1 PROCESSES
IO (DEV, PARM)
LEVEL 1 CPASCAL PROGRAM MONITORS, ETC.
CONCURRENT PREFIX
- - -
SEQUENTIAL PASCAL PROGRAM
SEQUENTIAL PREFIX
LEVEL 0 OS (NADEX) - MONITORS, MANAGERS, AND CLASSES
KERNEL

FIGURE 6

VIRTUAL CPASCAL MACHINE

initialization will be described at this point. This program structure is illustrated in Figure 7.

The SMM is the System Memory Manager. It manages dynamic NADEX system memory using a first-fit GETMAIN/FREEMAIN logic. The memory managed by SMM is allocated via the kernel GETMEM routine which returns a pointer to all unused memory within the task's region along with its size. Memory is managed using a descending-address ordered linked list of free blocks, where the first word is the link to the next free block and the second word is the length of this free block in bytes. All memory is allocated in 8 byte units. Dynamic memory is not obtained at the time the SMM is INITed, but is acquired later after all processes have been INITed (and thus have had storage allocated to them).

The next component is the SRM (System Resource Manager). This manager provides exclusive access ENQ/DEQ (enqueueing and dequeueing) control for arbitrarily named objects. Objects are identified by a pair of integers--the first denoting a class of resources and the second denoting a specific resource. The SRM is currently used for serializing system initialization, access to dynamic connect DTSs, access to non-shareable I/O devices (such as the simulated loader storage unit used to load the logon processor and the File Subsystem), access to shared DTSs (such as one IO node being used as a program loading source for multiple nodes), and serializing the loading of shared

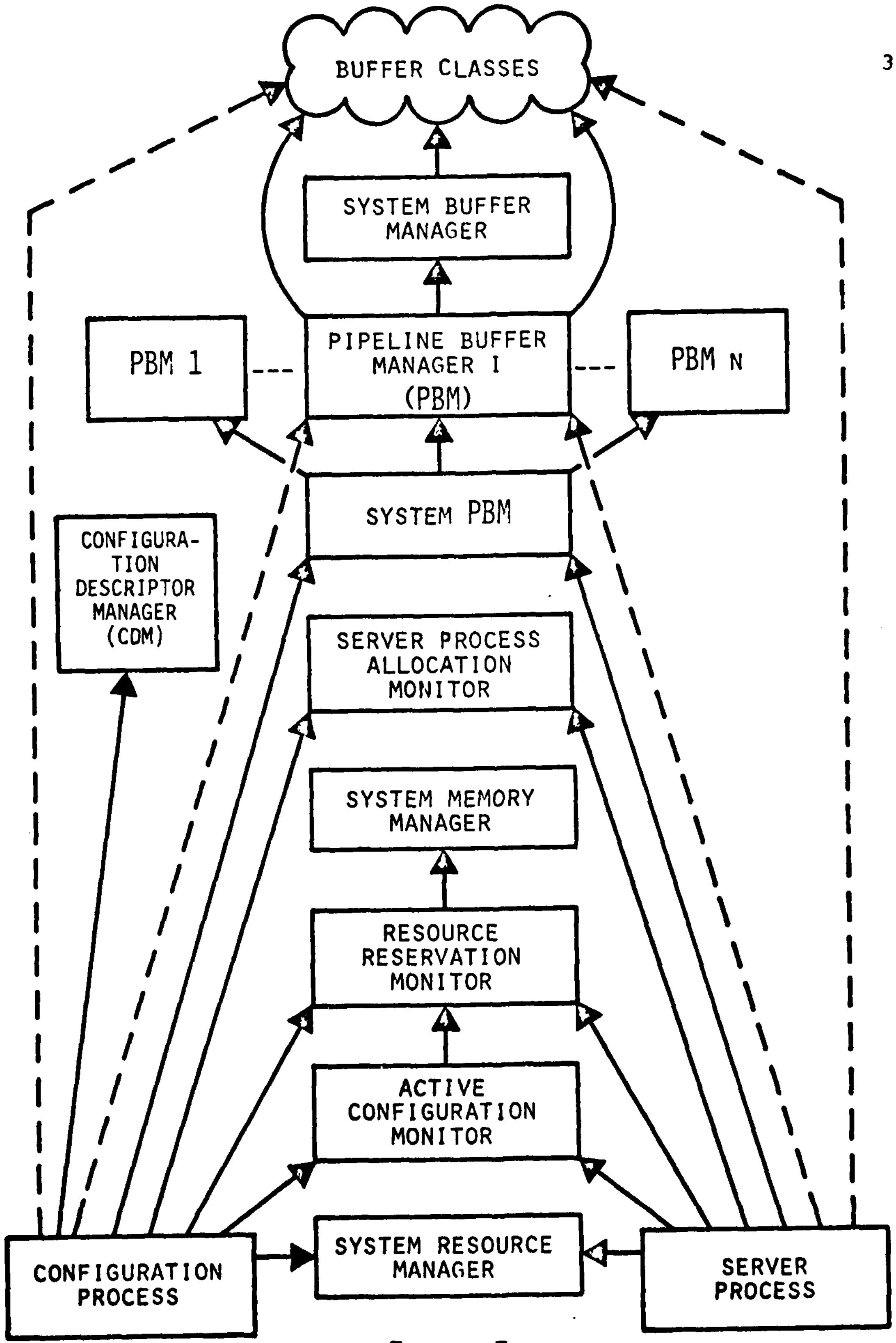

BUFFER CLASSES
SYSTEM BUFFER MANAGER
PBM 1
PIPELINE BUFFER MANAGER I (PBM)
PBM N
SYSTEM PBM
CONFIGURATION DESCRIPTOR MANAGER (CDM)
SERVER PROCESS ALLOCATION MONITOR
SYSTEM MEMORY MANAGER
RESOURCE RESERVATION MONITOR
ACTIVE CONFIGURATION MONITOR
CONFIGURATION PROCESS
SYSTEM RESOURCE MANAGER
SERVER PROCESS

FIGURE 7

NADEX IMPLEMENTATION (CONCURRENT PASCAL) programs.

The Resource Reservation Manager (RRM) performs allocation of system resources and controls access to shared programs. It contains a table RSRC_TABLE which contains the current allocation state of the resources it manages. The resources currently managed are system memory (in conjunction with the SMM), server processes, data and parameter buffers, configuration descriptors, and shared programs. The PROG_TABLE maintains the current state of shared programs including number of users, current memory location, program identification, and code space size. Initially, the tables are cleared and the resource tables (except memory) are initialized from the system generation constants which set the number of the various resources which are available. The RRM has access to the SMM in order to allocate/release memory.

The Active Configuration Monitor (ACM) maintains the status of all configurations in the system and is used for system-level inter-configuration communication. It also assigns configuration processes and provides access to the configuration descriptor while the configuration is active. Its CNFG_TABLE maintains the status of each configuration.

The SPAM is the Server Process Allocation Monitor. Idle server processes wait here, and it assigns server processes and activates them upon request from a configuration process. (Allocation is performed by the RRM.)

The SBM is the System Buffer Manager. The SBM contains the variable declarations for both parameter and data buffers, and performs assignment of buffers upon request. (Allocation is performed by the RRM.) It also contains the code for generating pointers to the buffers so that they can be passed to requesting processes.

The SPM is the System PBM Manager. The SPM contains variable declarations for the Pipeline Buffer Managers (PBMs). It will generate pointers to a specific PBM upon request. PBMs exist in a one-to-one correspondence to configuration processes, so no allocation is necessary and assignment is pre-defined.

The PBM is the Pipeline Buffer Manager. The name PBM is historical as the function is more appropriately called Configuration Buffer Manager or Configuration Resource Manager. The PBM is responsible for implementing DTS operations, as well as maintaining status information on the various nodes of a configuration.

The PBMI (PBM Interface) is a higher-level interface to the PBM which hides all of the multi-PBM aspects of subsystem access and which performs port to DTS mappings. PBMI's exist as variables of server processes.

The CDM is the Configuration Descriptor Manager. It assigns configuration descriptors similar to the way the SBM assigns buffers. Configuration Descriptors (CDs) are managed records which describe configurations which are active or which are to be activated at a later time.

Allocation of CDs is done by the RRM.

An SP is a server process. SP's are used to implement the nodes of a configuration which require an active entity (i.e., user programs, I/O access). A sysgen-defined number of SP's exist and are INITed during system initialization.

A CP is a Configuration Process. CP's are used to control configurations including the allocation of resources, configuration setup, configuration status monitoring, configuration termination, and freeing of resources. A sysgen-defined number of CP's exist also.

4.3 System Initialization

System initialization consists first of INITing all of the above, except the SP's and CP's. These components' initialization routines generally initialize their status tables and return. The initial process then ENQs on the system initialization resource by calling the SRM. It then INITs all of the SP's and CP's. The first statement of each of these is an ENQ on the system initialization resource, so they will all hang at that point. After all of the processes have been INITed, the RRM is called to cause the SMM to call the kernel to request (via GETMEM) the remaining memory in the task's region for use as dynamic system memory. At this point, all language-required allocation is complete since all processes have been INITed. System memory is initialized and its size placed in the RRM's entry for memory and control returns. The ENQ´s have prevented the CP´s and SP´s from executing to a point that they would have required system memory before it had been initialized. The initial process now DEQ´s the initialization resource. This allows the first queued process to acquire it, which immediately DEQ´s it. Thus, all of the processes are released to complete their initialization and begin processing.

A server process has no specific initialization other than INITing its PBMS. It then enters its processing loop, the first step of which (in routine INIT_NODE) is to call SPAM.SERVER_WAIT to wait for something to do.

A configuration process similarly enters its main processing loop in routine RUN. The first step is to call ACM.IDLE_CONFIG_PROCESS to wait for something to do.

The initial process then, to simulate the operation of the currently unimplemented Terminal Manager Subsystem, calls ACM.ALLOC_CONFIG_PROC to request allocation of a configuration process to terminals at logical units 16 and 19. Since there is no control over the timing relative to the initialization of the CP´s, this code must loop issuing WAIT´s and retrying until a CP is available.

4.4 User Signon

When a user attempts to signon to the system, that information is relayed by calling ACM.ALLOC_CONFIG_PROC passing the device id (logical unit number) of the user's terminal. In the case of the current system which does not include a terminal manager, this is done by the initial process, once for each terminal.

ACM.ALLOC_CONFIG_PROC finds an idle CP, updates its CNFG_TABLE entry to contain the terminal id, indicates that the configuration is in logon state, and schedules the CP to be continued. The CP, which was waiting in ACM.IDLE_CONFIG_PROC obtains the terminal id, userid (generated from the terminal id), and a CD pointer (null in this case since the CP was activated by a logon request).

The next step is for the CP to call the CDM to acquire a CD (since none was provided on the ACM call). The RRM is initialized with one CD allocated to each CP, so only assignment is required. The internal routine BUILD_LOGON_CD is called to build the CD describing the logon processor. This CD is hard-coded into NADEX and builds a 3-node configuration. The first runs the logon sequential program. The second is an I/O node for the console. The third is an I/O node for the permanently-assigned logical unit (emulated LSU) which contains the logon program code. The CD stack is initialized, and the CP prepares to run this CD. ACM.USER_LOGON is called to indicate that logon is complete. Currently, this merely changes the config state to CMDP and checks that a logoff request has not yet been received for the config. Processing then proceeds as for the general case of initiating a configuration.

4.5 Configuration Setup

The first step in setting up a configuration is calling ACM.CONFIG_ENTER_CMDP. This sets the current state to CMDP and verifies that a logoff request has not yet been received. The configuration id, terminal id, and user id are inserted into the CD. The COMPLETE_CD routine is called which verifies the correctness of the CD and builds some cross reference tables (such as CD_DTS_TABLE) within the CD.

If this is successful, ACM.CONFIG_ENTER_RRM is called to signify that the config is going to call the RRM. This also checks that an abort/logoff request has not been received. If that is successful, then RRM.RESERVE_CONFIG is called to allocate resources to the configuration.

The RRM is set up such that common processing may be performed for most resource types driven by calls to UPDATE_RSRC in routine UPDATE_CONFIG_RSRC. This routine is set up to allow backing-out of partially completed resource allocation in the event that complete allocation is not possible, as well as to handle the releasing of resources following completion of a configuration. Special handling is required for memory since the SMM must be called to ensure that a contiguous piece of memory of the requested size is available, and for programs since memory is required only if the requested program does not currently have memory allocated for it.

The basic logic is to allocate each resource in order from the CD until either all resources have been allocated or an error occurs. If all are allocated, control returns to the caller. If an error occurs, then all resources which were allocated are released. If it is possible that sufficient resources may become available eventually, the CP waits in the RRM. The particular resource on which the error occurred is saved so that the CP will be flagged as waiting for that resource; and whenever another process releases some of that resource, the CP will be awakened and the allocation process retried. Memory is actually assigned since only the SMM knows whether contiguous memory is available.

For programs, memory is allocated only if the program is non-shareable (not loaded from a subsystem) or if memory is not currently reserved for the program. Instead of returning the memory address in the CD, a program id is returned which will later be used to obtain the memory address from the RRM when the program is to be invoked.

After the resources have been allocated (and assigned in the case of memory and programs), the PBM associated with this configuration is acquired from the SPM. PBM.INIT_CONFIG is called to initialize the PBM with information from the CD about the configuration which is to be run. The CD contains information about the number of nodes, number of DTS's, DTS and connection characteristics, etc. The PBM calls the SBM to assign the specified numbers of parm and data buffers.

The CP then calls ACM.CONNECT_SUBSYS to request activation of any subsystems used by this configuration. If the requested subsystems are already running, their user count is incremented and no other processing is required. If they are not running at all, a CP is assigned to run the subsystem and it is continued (from its call to ACM.IDLE_CONFIG_PROC). In this case, it will receive a null terminal id and the subsystem name will be the user id. As above, it will build a CD to run logon which will build the CD to run the subsystem. The user's CP will wait in the ACM until the subsystem has been activated (not counting the logon processor). If the subsystem is changing states, the user waits until the state change is complete and then determines whether it can be used or whether the subsystem must be activated (i.e., it just terminated).

When ACM.CONNECT_SUBSYS returns with a successful return code, it means that all of the required subsystems are running and their user counts have been incremented. The subsystem names are obtained from the CD.

Local routine (in the CP) CONNECT_SUBSYS is then called to perform the dynamic connection to each subsystem. Dynamic connection consists of writing a parameter to the subsystem over its DC DTS requesting assignment of a user interface DTS for this user. A CD-supplied parameter may also be written to define the function of the DTS. The subsystem then writes a parameter which contains the assigned DTS id which is placed in the CD. All of this is done under an ENQ for the subsystem's DC DTS in the SRM to serialize access to the DC DTS. In the event of an error response from the subsystem, the user's PBM is informed by a simulated disconnect call that the DTS is not usable.

The CP then in routine START_NODES calls SPAM.ASSIGN_SERVER to assign a server process to each node requiring one. Due to sequencing constraints, the CP may have to wait for an SP to be available (this is only a transient condition since the RRM guarantees that one will be available). The pointer to the CD is then passed through the SPAM to the server process which uses it to perform its initialization and then passes it back to the CP through the SPAM. The CP is continued at this time and returns.

The CP then performs the configuration status monitoring function.

4.6 Server Process Operation

The normal processing loop for a server process begins with a call to SPAM.IDLE_SERVER (from local routine INIT_NODE). This routine returns the pointer to the CD for the config, the config id, and the node id for this particular node. The SP extracts information pertaining to its node from the appropriate fields in the CD into its own variables.

Before returning the CD, the server calls PBMI.INIT_NODE passing in the CD. Note that the PBMI is a class which is a variable of the SP. PBMI.INIT_NODE performs some initialization functions. It clears its PBM and port tables. A pointer to the local PBM is obtained from the SPM. The INIT_NODE routine of the local PBM is called which saves the kernel´s process id (for use in a STOP request) for the SP and checks if an abort has already been requested for this node. Upon return to the PBMI, the port table is then built using information for this node from the CD. The PBM´s MAP_DTS routine is used to map CD DTS ids into PBM DTS ids. For connections to a subsystem, routine ADD_PBM is called to obtain a reference from the SPM to the subsystem PBM and add it to the PBM table.

When PBMI initialiation is complete, it returns to INIT_NODE in the SP which calls SPAM.RELEASE_CDP to return the CD to the CP through the SPAM. INIT_NODE then returns to the main processing loop of the SP in routine RUN.

The SP then performs the appropriate processing for the node. For an IO node, a device-type-dependent processing routine is called (CONSOL_IO, SEQL_INPUT, SEQL_OUTPUT). For a user program (sequential or concurrent), the SRM is called to ENQ on the program loader resource for the specified program id. RRM.FETCH_PROG is called to obtain a pointer to the memory assigned to the program and to determine whether the program is already loaded (from the status in the RRM´s PROG_TABLE). If it is not already loaded (this is the first user or the program is private), then local routine LOAD_PROGRAM is called to load (and relocate) the program into the specified memory over the program loader port specified in the CD. Otherwise, the program has already been loaded and relocated and is ready for execution. The program id is DEQed in the SRM and the program is invoked with the requested prefix (native or PASDRIVR-compatibility).

Program loading is performed by reading pages over the program loader port and copying them into the assigned memory. The RELOC_PROGRAM routine is called to relocate adcons and RX3 instructions using RLD information loaded into memory along with the program. Note that access to this memory during the load/relocate process is serialized by use of the SRM and that no language protection is provided. This occurs because the program memory is effectively a managed record to which there are multiple pointers.

Since DTS operations will be discussed elsewhere, the discussion of those routines in the native pefix and the routines of the PASDRIVR prefix which map into DTS operations will be deferred. The native prefix contains some other routines, however, which require special handling by NADEX.

The FETCH_USER_ATTR native prefix routine returns information from the SP's variables about the configuration and node, such as user id, terminal id, and node number. The SVC1 and SVC7 prefix routines merely invoke the kernel's SVC1 and SVC7 functions. Note that these uses are currently unprotected and that no facility exists in the current RRM for assignment of LU's. FETCH_PARM allows the user to obtain the parameter information for the program which was saved from the CD one parameter buffer at a time.

Invocation of overlay programs is done in two steps through the native prefix. The first is the loading of an overlay program done through LOAD_OVERLAY. The user provides the port number of a port which he has pre-configured such that the program loader can just read pages of code into memory from the DTS until end of file is detected. The program is relocated after it is loaded. The size of the overlay area was specified in the CD and the overlay memory was passed to the SP in the CD from the RRM. The second step is the INVOKE_OVERLAY routine which actually invokes the program with a user-supplied parameter and returns the program result when the program terminates. Note that NADEX does not currenty support multiple levels of overlay since only one overlay area exists and NADEX does not have information to be used in restoring the previous contents of an overlay area.

The CANCEL_NODE routine may be used to request that the node be cancelled. That will be discussed in the section on node termination. CANCEL_CONFIG is used by special authorized users to request the ACM to cancel other configurations.

The SUBMIT_CONFIG prefix routine is used to allow a user program (such as the logon processor or a command processor) to present a new CD for execution. Execution may be in one of three modes. CALL and XCTL take effect when the current configuration terminates. CALL indicates that the new config is to be executed, and when it terminates, the current config is to be re-invoked. This would be used by a command processor when it invoked a user program, for example. XCTL indicates that the new config is to be executed in place of the current one. This would be used by the logon processor when it transfers control to a subsystem or a command processor configuration, since return to the logon processor is not required. The third type is a SPIN_OFF which means that the new config is to execute asynchronously.

All SUBMIT_CONFIG requests go to ACM.SUBMIT_CHECK to check whether the user has CD's allocated to perform the requested function and whether the user is violating the rules on submits (e.g. more than one of type CALL/XCTL). The CNFG_TABLE in the ACM keeps track of how many CD's have been used from the config's allocation. If the submit is allowed, the CDM is called to acquire a CD and the CD is copied from the user's parameter into the CD. ACM.COMPLETE_SUBMIT is then called to complete the submit operation. For XCTL or CALL type, the CD pointer is saved in the CNFG_TABLE so that it may be passed to the CP along with the submit type. For a SPIN_OFF type, the ACM attempts to allocate an idle CP (waiting in ACM.IDLE_CONFIG_PROC) passing it the CD for the config it is to run. If this is successful, that CP will be entered with a non-nil CD pointer so it will skip the building of a CD for the logon processor and go directly to config setup for the specified config. If the spin-off was unsuccessful, the CDM is called to release the CD. The spun-off CD no longer counts against the config's CD allocation since it is effectively replaced by the pre-allocated CD for the CP which is running the spun-off config.

4.7 User Initialization

To put this in perspective, let us look at how it is used by the system when a user logs on. The logon processor seeing that it is not running on behalf of a subsystem builds a CD for the system command processor and submits it with the XCTL. It then terminates and the command processor is invoked in its place. Since the command processor requires the File Subsystem (FSS), another config will be activated to start up the FSS if it is not running. This config will also invoke the logon processor which will build a CD to run the FSS (loading it from a pre-assigned logical unit). It then submits it with type XCTL and terminates which causes the FSS to be activated. When the FSS is running, this allows the command processor to be activated. The command processor communicates with the user and the FSS and eventually builds a CD to run the user's configuration. It submits it with type CALL and terminates which causes the user's configuration to be run. When the user's configuration is finished, the command processor configuration is re-invoked. When the user issues the SIGNOFF command to the command processor, the command processor configuration terminates; and since there is no configuration to re-invoke, the user is logged off and the CP returns to the ACM to await reassignment.

4.6 Server Process I/O

The SEQL_INPUT routine of the server process merely reads pages from the specified logical unit using SVC1 and writes them to port 1 of the node. The routine ENQ's on the logical unit in the SRM before beginning so that exclusive access is obtained (since sequential reads are used). When end-of-file is reached, the logical unit is rewound and data transfer continues. (An end-of-file buffer is sent to the user, however.) Transfer stops when an error status is obtained on the write to port 1. At that time, the logical unit is DEQed and the routine terminates.

The SEQL_OUTPUT routine, when implemented, will perform a similar function for output files. Currently, SEQL_INPUT is used only for program loading from pre-assigned logical units for the logon processor and the file subsystem, and no use has been found for the output function.

The CONSOLE_IO routine performs interactive I/O to a terminal device using SVC1. The console protocol is defined elsewhere and will not be repeated here other than noting that the console is initially in write mode and that writing an EM requests a read of one line.

4.9 Normal Server Termination

A server normally terminates by returning to procedure RUN which then calls TERMINATE_NODE. This node then calls RRM.RELEASE_PROGRAM to release the program (if any) and calls RRM.RELEASE_MEMORY to release the memory for the data space and overlay code space for the node. Memory is released in this manner since there is no easy way to obtain access to the CD to place the memory pointers back in it so they may be passed to the RRM by the CP. Also, experience has proven that memory is the most scarce resource and thus releasing it as soon as possible (each node's termination rather than config termination) may be beneficial.

The server then calls PBMI.TERMINATE_NODE. This first calls the TERMINATE_NODE routine in the local PBM which flags the node as aborted and aborts all of the appropriate DTSs which cannot be used without this node. Then for each port, any buffers held in the PBMI (due to character operations) are returned to the appropriate PBM; and the port is disconnected by the DISC_PORT routine. Finally, the PBMI returns to the SP.

The SP then calls ACM.TERMINATE_SERVER to inform the ACM that a server process of this config has terminated.

The ACM keeps track of the number of active servers and, when it becomes zero, informs the CP that the config has terminated. The SP then returns to the top of its processing loop which is to call SPAM.SERVER_WAIT to wait for reassignment.

4.10 Abnormal Server Termination

Abnormal termination can be initiated from several places. Various error checks in the prefix routines of the server itself may request termination. The PBMI may request termination due to bad responses from the PBM. The PBM may request it due to some error condition. Finally, termination may be requested by some other process (like a system operator).

External aborts are handled by requesting an abort by calling ACM.CANCEL_CONFIG. (Note that externally you can cancel only an entire configuration.) The abort request and its level (abort config or logoff user) is stored in the CNFG_TABLE; and if this is a stronger request than one which is pending, the CP is continued. The CP will have been waiting at ACM.AWAIT_CONFIT_EVENT and will receive a return code which causes it to call ABORT_CONFIG in the local PBM.

The PBM ABORT_CONFIG routine merely calls ABORT_NODE for all of the nodes. ABORT_NODE flags the node as aborted and saves the abort reason (typically a line number from the original requestor). If the node has called INIT_NODE, then the kernel's process id is available and a STOP is done on the process. If the node is waiting in the PBM, then it is continued and all of the DTSs associated with the node are aborted.

An ABORT requested in the PBMI causes the PBMI's ABORT_REQUEST flag to be set and causes the PBMI to stop the server. An ABORT in the server calls PBMI.ABORT which also causes the stop.

The STOP causes the kernel to pop user program levels whenever they would be executed. However, prefix execution will not be interrupted. Thus, the prefix routines need only check explicitly for abort conditions in places where they can loop or can enter a wait condition. The I/O routines check the PBMI status codes for a REQ_ABORT_NODE status which indicates that the node has been aborted in the PBM or PBMI. Routines such as the program loader which use DTS operations also make such checks. Thus, the node will fairly quickly return to the point in the processing loop where it performs the normal termination procedure.

4.11 Configuration Status Monitoring

While the configuration is executing, the CP is waiting in ACM.AWAIT_CONFIG_EVENT. The events of interest are quiesce request (for non-permanent subsystems), abort request, logoff request, and configuration termination (i.e., all servers have terminated). For a quiesce request, the CP writes a quiesce-request parameter over the DC DTS to the subsystem it is running. For an abort or logoff request, ABORT_CONFIG in the local PBM is called to initiate configuration abort. For these cases, ACM.AWAIT_CONFIG_EVENT is called again to wait for another event. Each of these events will be reflected only once, and only if a higher precedence event has not already been reflected. The ACM keeps track of this using CNFG_STATE, which is the state reflected to the CP, and CNFG_NEW_STATE, which is the state requested (e.g., ACTIVE, QUIESCE, ABORT, LOGOFF). A quiesce is requested only for a non-permanent subsystem when its active user count reaches zero.

4.12 Configuration Termination

When all of the servers have terminated (the count is maintained in the ACM and decremented when ACM.TERMINATE_SERVER is called), then the termination event is returned to the CP on the ACM.AWAIT_CONFIG_EVENT call. This breaks it out of the loop calling ACM.AWAIT_CONFIG_EVENT.

ACM.DISCONNECT_SUBSYS is then called (with the CD) to disconnect from all of the subsystems to which the config was connected. Disconnecting decrements the active user count and causes non-permanent subsystem with active user count of zero to be signaled to quiesce. Note that since all of the servers have terminated, all of the PBMI's have completed the DTS disconnects from the subsystem PBM's.

ACM.CONFIG_CHECK_STATE is called to determine whether a logoff was requested. The RELEASE_BUFFERS routine in the local PBM is called to return allocated buffers to the SBM. RRM.RELEASE_CONFIG is called to release all resources (except memory and programs) allocated to the config.

Local routine UPDATE_CD_STACK is called to handle the CD stack used by the SUBMIT_CONFIG prefix routine. If no CD's were submitted, the top entry is popped from the stack. If a CALL type submit was issued, the current CD is pushed onto the stack and the submitted CD becomes the current CD. If an XCTL type submit was issued, the current CD is replaced by the submitted CD. The submit information is obtained from ACM.FETCH_SUBMIT_CD. Any CDs allocated to the config that were not used are released by calling RRM.RELEASE_CDS. If this was a subsystem and no submit was performed, ACM.SUBSYS_RESTART_CHECK is called to determine whether the subsystem should be rstarted (e.g., there is a user already waiting for it). If that is the case, then the current CD becomes the new CD. When CD's are popped or replaced, the old CD is released by calling the CDM and the allocation is released by calling the RRM. If the stack is empty when a pop is required, then the user will be logged off.

If a logoff was not requested (either by the ACM or because a pop from the CD stack was required and the stack was empty), then the CP loops back to setup the new configuration. Otherwise, the user will be logged off. The CD stack is emptied, freeing all of the CD's in the CDM and RRM. The CP then repeats its processing loop by calling ACM.IDLE_CONFIG_PROC.

5.0 Bibliography

1. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, 1977.

2. Holt, R. C.; Graham, G. S.; Lazowshka, E. D. and Scott, M. A., Announcing Concurrent SP/K, Operating System Review, 12, 2 (April 1978).

3. Brinch Hansen, P., Operating Systems Principles, Prentice Hall, 1973.

4. Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," CACM, 17, 10 (October 1974).

5. Dowson, M, The DEMOS Multiple Processor Technical Summary, National Physical Laboratory Technical Report, NPL Report 101, April, 1978, Teddington, Middlesex TWII OLW, UK.

6. Hoare, C. A. R, Communicating Sequential Processes, CACM, Vol. 21, No. 8, (August) 1978, pp. 666-677.

7. Farber, D. J., et al., "The Distributed Computing System Digest of Papers," COMPCON 73, February 1973, pp. 31-34.

8. Digital Equipment Corp., VAX-11/780 Software Handbook, 1977.

9. Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, 1972.

10. Thompson, K. and Ritchie, D. M., The UNIX Time-sharing System, CACM, Vol. 17, No. 7, July 1974, pp. 365-375.

11. Ritchie, D. M., A Retrospective in UNIX Time-sharing System, The Bell System Technical Journal, Vol. 57, No. 6, Pt. 2, (July - August), 1978.

12. Brinch Hansen, P., The SOLO Operating System, Software Practice and Experience, Vol: 6, No. 2, April - June 1976, pp. 141-206.

13. Sunshine, C., Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations,

Rand Tech. Report R-2064/1-AF, June 1977.

14. Zucker, S., Interprocess Communication Extensions for the UNIX Operating System: II. Implementation, Rand Tech. Report R-2064/2-AF, June 1977.

15. Rochat, K.; Wallentine, V.; and Young, R., A Software Configuration Control System, (in preparation).

16. Morrison, J. P., Data Stream Linkage Mechanism, IBM Systems Journal, Vol. 17, No. 4, 1978.

17. Young, R. and Wallentine, V., The NADEX Core Operating System Services, Dept. of Computer Science, Kansas State University, Technical Report, TR-79-11, November 1979.

18. Young, R. and Wallentine, V., The Kernel of CPascal/32, Dept. of Computer Science, Kansas State University, Technical Report, TR-79-13, December 1979.

19. Silberschatz, A.; Kieburtz, R. B.; and Bernstein, A. J., "Extending Concurrent Pascal to Allow Dynamic Resource Management," _IEE Trans. Software Eng._, Vol. SE-3, pp. 210-217, May 1977.

DATE
FILMED
0-8