Preliminary Design Of A Port-Oriented Operating System

David Stemple
Steve Vinter
Krithi Ramamritham

82-24

Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

1. Introduction

In this report we describe a preliminary design of a port-oriented operating system. We will call this system the Gutenberg system because all port activity is driven by moveable type definitions placed in a directory by the user. Throughout we adopt the abstract data type approach to specification and implementation and use the term _object_ to denote an instance of an abstract data type. We see the kernel as a type manager for the abstract data type port. The capability manager, which builds a hierarchical type directory and controls processes' port capabilities, is seen as the manager of the directory object. In addition, we view processes as either managers for object types passed on ports (the port server processes) or as requesters of operations on the object types (the port owner processes). Even processes themselves are seen as objects to be passed on ports, since the port managing kernel uses itself recursively to manage processes.

The Gutenberg system consists of ports, objects sent on ports, processes, and a capability directory. Processes may request the creation of ports and then execute operations which send and receive objects on the ports. Sending and receiving objects on ports leads to the creation, blocking, and starting of processes, and is the only mechanism controlling processes after the system is initialized. The _capability directory_ contains the information used to control the creation and use of ports, and also determines the process images which are executed when processes are created. The capabilities management scheme outlined in this report is a rudimentary password-based technique which we intend to expand and improve.

This report presents a logical design of the operating system kernel. Details of hardware context switching, interrupts, memory management, process priorities, and IO device handling are not included. The report is organized as follows. The next section gives an overview of the Gutenberg operations. An example of port usage in file I/O is then presented. The fourth section presents detailed descriptions of the port operations, their purposes, and their parameters. This is followed by a description of the capability directory and its operations. The next section introduces another example to illustrate the use of port capabilities to achieve a protected abstract data type view of a database and its operations. Then process management by the port manager (the operating system kernel) is outlined.

2. Overview of Gutenberg Operations

In the Gutenberg system we regard port as a generic abstract data type. Accordingly, a port is defined as an object which can only be operated on by the operations: Createport, Send, Receive, Selectreceive, Acceptrequest, Waitany, Waitall, and Destroyport. The semantics of these operations are defined generically, i. e., without specifying the types of the objects to be sent or received. The port creation operation, Createport, not only creates a port for a process, but also associates the port with a specific type of object which may be sent or received on it. Subsequently, to allow a finer granularity in the granting of capabilities, we wish to associate ports with specific operations on the objects.

Objects to be sent or received on a port are typed within the context of an operating system directory structure which exists outside of any process. This typing is separate from whatever typing of objects is done within the process. Such internal/external typing is not unlike that which occurs with records which are typed inside a program one way and typed, perhaps differently, as elements of a database. However, as we will see, objects sent on ports are actually uninterpreted objects to the operating system and include file records, process images, processes, and resources.

In the Gutenberg system, processes which either wish to send or receive objects of some type from (to) another process establish a port for that purpose. A process may create a port either for sending or receiving, but not both. The process creating a port is the _owner_ of the port. Port ownership rights include the right to destroy a port and wait for multiple service requests on the port. When a port is created, the owner must specify the type of objects to be passed on the port. The named object type must be in the subdirectory of the capability directory (similar to a directory node in a UNIX file directory) pointed to by the process requesting the port allocation, i.e., the owner. This subdirectory is denoted the _active directory_ of the owner. Generally, processes are capable of changing their active directory to different subdirectories nodes within the capability directory throughout their existence.

An object type name in a directory is linked to at least one process image. A link from an object type name to a process image carries one of three labels: "R", "SR", or "S". The meaning of the link and its label, say an "R", is that a process using the directory can establish a port on which to Receive (Send if "S", Selectreceive if "SR") objects of the type. The link, called a capability link, denotes the <u>privilege</u> of a process using the directory to create a port for use with the specified operation. A process image connected to an object type by an "R" or "SR" link can be thought of as a producer of the object type. A link labeled with "S" marks the process image as an inventory manager or consumer for the object type. When the owner process executes its first Send, Receive, or Selectreceive on a port, the process image associated with port's object type is used to create a process that will also communicate on the port. This process is called the <u>server</u> of the port.

We now give a brief overview of the Gutenberg operations. The Createport operation causes a port to be created for use in passing objects of a particular type (given in the arguments). The calling process is established as the owner of the port. The Createport parameters include an operation mnemomic which specifies the only send or receive operation which may be executed on the port by the its owner. These operation mnemonics must match the labels on the capability links. An error status is returned if the object is not in the directory currently used by the process or the privilege is the specified operation is not declared in the directory.

The Receive operation requests an object on a port. If one or more objects are queued on the port, the first one is delivered and control is passed to the next statement after the Receive. If the port is empty, and the waitnowait parameter of the call is "wait", the calling process is blocked until the sender attached to the port sends an object. When an object is sent on a port and the receiver of the port is blocked, the receiver is unblocked and the object is delivered.

The Send operation causes an object to be queued on a port and may unblock a receiving process. If the port is full, the sending process is blocked until an object is removed from the port

The Selectreceive operation is used to receive objects selectively. The operation's inputs include information to be used by the port server in choosing the object to be returned on the port. Since the selection information is sent to the server on the port and the server sends back the requested object on the port, the port is said to be <u>bidirectional</u>. Thus a process may request a particular object of a type by using a bidirectional port send the details to be used in choosing the object.

The Acceptrequest operation is issued only by a port server. It is issued to determine the port on which to send or receive objects. Acceptrequest also returns the object type name associated with the port, and, in the case of bidirectional ports, returns details of the request.

The Createdir, Register, Changedir, and Removeport operations involve the capability directory, not ports. Createdir adds a subdirectory node to the current subdirectory. Register allows an object type name to be placed in the capability directory. Changedir changes the capability directory node currently associated with the calling process to a new node, and thereby changes the capability of the process to create ports. Removeport provides the ability to delete object types from the directory.

3. An Example of Gutenberg Port Usage

Let us consider a simple example of ports used for file (or more precisely, record) input. Assume process A is running and needs to read a record of type PERSON-REC. Before the process is started, a directory structure, which we call a capability directory, must have been built. A running process always points to a node in this directory, its active directory. The node defines the process' capability for creating ports. In our example, the object type PERSON-REC must occur in the active directory of Process A; i. e., the USERDIR subdirectory. The following diagram illustrates the situation.

```
      Processes                   Capabilities Directory

    +-----------+
    | Process   |------------------> USERDIR
    |    A      |                       |
    +-----------+                       |
                                        |
                                    PERSON-REC     FILEMGR
                                        |             ^
                                        |             |
                                        |      R      |
                                        +-------------+
```

System before Createport

```
      Processes                    Capabilities Directory

     +-----------+
     |  Process  |-------------------> USERDIR
     |     A     |                     ^  |
     +-----------+                     |  |
           ^              +------------+  |
           |              |               |
           |              |               |
           | port         |           PERSON-REC   FILEMGR
           |              |               |             ^
     +-----------+        |               |             |
     |  FILEMGR  |--------+               |      R      |
     |  Process  |                        +-------------+
     +-----------+
```

System after Createport and Receive by Process A

By being attached to the capability directory node labelled USERDIR (a subdirectory node), process A has the capability for creating ports for the purpose of receiving objects of type PERSON-REC. This is shown by three arrows, the first from the process to the directory node USERDIR, the second from USERDIR to PERSON-REC, and the third labelled with R (receive privilege) from PERSON-REC to the process image node labelled FILEMGR. FILEMGR, like PERSON-REC, is in the capability directory and is located under some subdirectory node not shown here. The capability directory is organized in a manner similar to a UNIX file directory, with USERDIR corresponding to a directory node and FILEMGR to an executable file.

When Process A executes a Createport, a port is created by the operating system kernel. Process A, the port owner, is then marked (in the operating system's process list) as having destroy, wait, and receive capabilities on the port and the port identifier is passed to

Process A. When process A executes the first Receive on the port, a new process is initiated using the process image FILEMGR. This process is given Acceptrequest and Send capabilities on the port. Using the Acceptrequest operation, the FILEMGR process finds out what request has been made, namely a Receive of an object of type PERSON-REC on this particular port. FILEMGR then gets a record from the appropriate file and Sends it on the port. Process A is unblocked by the Send and is given control at the point after the Receive, the record having been placed in Process A's buffer as specified in the Receive. Subsequently, FILEMGR executes another Acceptrequest which blocks itself until another Receive on the port is executed.

Of course this is a simple example and illustrates only one use of ports and their capabilities. It should be noted that Process A does not specify, in this scheme, the name or identity of the process which will ultimately satisfy the Receive request. The fact that PERSON-REC type objects are produced by FILEMGR is only bound to a port at port creation time and then from information in the capability directory, not from information supplied by the process. The program view (Process A's) of the port is that it is used for Receiving objects of type PERSON-REC. Nowhere in the program is it specified that FILEMGR is to be the sender at the other end of the port. If subsequently it is required to get PERSON-REC records from another program, say a user written program, a query processor, or a program on another node in a network, then the PERSON-REC object type would only need to be linked to another process image node, either the compiled user program, the query processor, or the network manager. These programs would, of course, have to be capable of receiving and servicing requests for PERSON-REC objects. In the case of a query processor, it would be necessary for it to have its own directory of defined queries (or views), one of which could be used to produce PERSON-REC objects.

From the above we can see that when a port is created it is made an instance of a specific abstract type, i. e., an instance of the type of ports on which a particular object type may be sent. The specific port type is specified in much the same way as a generic stack type is made into a specific integer stack subtype by further specification. Here, the further specification is a part of the Createport execution, for it is at this point that the port is first associated with the type of objects to be sent on it.

One of the features of this system and the use of abstract data types in general is the ability to hide details of operation implementation. In the example, Receive is implemented as a read of a record from a file. Not only is this fact hidden from the receiving process, but the operating system kernel itself has access only to the name of the implementation (FILEMGR) not to the implementation details. Consistent with other examples of abstract data types, the hiding of implementation details allows us to choose different implementations, as illustrated in the discussion of alternate producers of the PERSON-REC records.

The high level of the port operations and the ability to use the capability directory to bind the Receive and Send operations to different implementations (process images) allows us to implement diverse functions in a flexible but protected manner.

4. Gutenberg Port Operations

In this section we present descriptions of the primitive port operations of the Gutenberg system.

CREATEPORT (portid,objectype,operation,status)

Executed by: A process desiring to create a port. The calling process becomes the owner of the port.

Purpose: To establish a communication link between the owner process and the server process. The created port is bound to the specified object type and the creating process obtains Destroyport and Wait capabilities for the port along with one other:

Send if <operation> is 'S', port will be undirectional.
Receive if <operation> is 'R', port will be unidirectional.
Selectreceive if operation is 'SR', port will be bidirectional.

Parameters:

<portid>: (output) Value of port identification of the established port.

<objectype>: (input) Name of type of object to be passed on port.

<operation>: (input) Indication of how owner will use port.
'S': Owner is the sender of objects on the port.
'R': Owner is the receiver, using Receive, of objects on the port.
'SR': Owner is the receiver, using Selectreceive, of objects on the port.

<status>: (output) Indication of the result of the port creation attempt.
0: Successful port creation
1: Error in calling sequence; illegal parameter.
2: Error - object type is illegal.
3: Error - object type is unavailable; owner does not have privilege over object type.
4: Error - Too many ports allocated.

RECEIVE (portid,buffer,waitnowait,status)

Executed by: The owner if Createport parameter <operation> = 'R', or by the server if Createport parameter <operation> = 'S'

Purpose: To receive an object of type <objectype> on port <portid> from another process.

Parameters:

<portid>: (input) Port identification from which object is to be received.

<buffer>: (input) Location to place received object.

<waitnowait>: (input) If 'wait' then process is blocked until a SEND is performed on the port. If 'nowait' and no objects are available on the port, the process executing the Receive continues, status will be set to 4 and will be reset by the port manager when the Receive is completed. See Waitany and Waitall below.

<status>: (output) Indication of the result of the attempted receive.
   0: Successful receive performed.
   1: Error in calling sequence; illegal parameter.
   2: Error - illegal portid/objectype pair.
   3: Error - Receive not preceded by server's Acceptrequest.
   4: Receive pending, no error. Returned only if <waitnowait> = 'nowait'.

SELECTRECEIVE (portid,buffer,requestdetails,waitnowait,status)

This operation is the same as Receive except that, in addition to Receive's actions, request details are sent on the bidirectional port identified by portid.

SEND (portid,buffer,status)

Executed by: The owner if Createport parameter <operation> = 'S', or by the server if Createport parameter <operation> = 'R' or 'SR'

Purpose: To send an object on port identified by <portid> to another process.

Parameters:

<portid>: (input) Identification for port on which the object is to be sent.

<buffer>: (input) Location of object to send.

<status>: (output) Indication of result of the attempted Send.
   0: Successful Send performed
   1: Error in calling sequence; illegal parameter.
   2: Error - illegal portid/objectype pair.

3: Error - Send not preceded by Acceptrequest by server.

ACCEPTREQUEST (objectype,portid,operation,directionality,
requestdetails,waitnowait,status)

Executed by: A port server.

Purpose: 1) To receive information for the next request to be serviced.
2) To poll its ports for pending requests to be serviced (if <waitnowait> = 'nowait')
3) To receive details of a request in <requestdetails> (if Createport called with <directionality> = 'B')

Parameters:

<objectype>: (output) Object type of pending request.

<portid>: (output) Port identification for port on which request was executed.

<operation>: (output) Operation mnemonic (see Createport) of owner's request

<requestdetails>: (output) Details of service request. Set only if port is bidirectional.

<directionality>: (output) Indication of the directionality of the port returned:
'U': port is unidirectional
'B': port is bidirectional - <requestdetails> have been returned.

<waitnowait>: (input) Determines whether process should be blocked if no request is pending:
'wait': block process if no request is pending;
'nowait': return with <status> = error if no request is pending.

<status>: (output) Indication of result of <acceptrequest> attempt.
0: Successful <acceptrequest> performed.
1: Error in calling sequence - illegal parameter.
2: Error - <waitnowait> = 'nowait' and no request is pending

WAITALL (portlist,status)

Executed by: The owner or server of the ports in portlist.

Purpose: To suspend the calling process until pending nowait Receive and Selectreceive operations on all ports in portlist are completed.

WAITANY (portlist,portid,status)

Executed by: The owner or server of ports in portlist.

Purpose: To suspend the calling process until any pending Receive or Selectreceive operation on a port in portlist is completed. The parameter portid is set to the port identifier of the port of the completed operation. If more than one operation has completed, portid is set to the port of one of the operations arbitrarily.

DESTROYPORT (portid,status)

Executed by: The port owner.

Purpose: To destroy the port identified by portid.

5. The Capabilities Directory

The capability directory contains linked subdirectories which determine the port capabilities in terms of port operations, object types, and process images. An object type is represented by a node in the capability directory. Each object type node is connected to process image nodes by capabilities links labelled with port operation mnemonics. The meaning of the object type is determined by the process images to which its name is connected in the capability directory, since the process images are the implementations of the port operations labelling the capability links. Thus, in the example above we know that PERSON-REC is a record type since it is connected with the file manager. Such a view is in concert with certain approaches to abstract data types in which the meaning of the data is contained wholly in the permitted operations on the type.

The primary function of the capability directory is to control the creation of ports. Each running process is associated with a subdirectory node, its active directory, in the capability directory. Subdirectory nodes are object types like any other nodes, and are known to be subdirectory nodes by being attached to the capabilities manager process image. Subdirectory nodes are different in one respect from other nodes in that they can have unlabelled connections to the object type names which are said to be "in" the subdirectory. These connections in the active directory of a process determine the _view_ of the process by defining the process' capabilities for creating and subsequently operating on ports.

Names of process images are also nodes in the capability directory and have an object type interpretation determined by the capability links. By convention a process image has one capability link connected to itself and labelled 'SR' for Selectreceive. We may think of a process image as an object type, instances of which are its individual executions. The object produced by the process image to "objectify" its execution instance is its termination message.

Normally one process initiates another process indirectly by operating on a port associated with an object type connected to the second process' image. We saw this in the example above in which a file manager process is initiated by a Receive of a PERSON-REC object. However, a process can directly initiate a second process if its process image name is in the running process' active directory. To do this the first process creates a port for purposes of executing Selectreceive, and provides process image name as the object type associated with the port. We will call a port created in such a manner an execute port. After creating the execute port, the port owner executes a Selectreceive on it to initiate the second process. The two processes will be connected by the created port. At termination the second process executes a Send of its termination message on the execution port. This notifies the owner of the port that the second process has terminated.

In order for the first process to communicate its desires to the second process it can send information, including ports, in the request details of the Selectreceive. For example, the terminal command processor, in order to execute an interactive editor, would send four ports to the editor process -- the terminal input and output ports, a port for reading the file to be edited, and a port to use in writing the edited output. Upon termination these ports would be returned to the command processor.

The capability directory is rooted at one node and is organized in a manner similar to a UNIX file directory. The immediate successors of the root node are the system initialization process images, and the users' top level subdirectory nodes. Below the user subdirectory nodes are the users' object types, including process images and subdirectories. To each subdirectory node is attached protection information in the form of passwords paired with the capability directory operations, Createdir, Changedir, Register, and Removetype.

In the next section we give descriptions of these operations and how the passwords are used to govern their use.

6. Capabilities Directory Operations

CREATEDIR (dir-name,password-operation-list,password,status)

Executed by: Any process.

Purpose: To add a subdirectory node to the object types of the current subdirectory of the calling process.

Parameters:

<dir-name>: (input) The name of the new subdirectory.

<password-operation-list>: (input) List of password-operation pairs used to establish the capability to execute the operations on the new subdirectory. The operations can be Createdir, Register, Removetype, and Changedir.

<password>: (input) Password used to gain ability to execute a Createdir on the current subdirectory.

<status>: (output) Indication of the result of the Createdir operation
0: Successful Createdir performed
1: Failed to perform Createdir due to insufficient privilege

REGISTER (objectype,processimage,operation,password1
password2,status)

Executed by: Any process.

Purpose: 1) To enter an object type name into the capability directory under the current directory node of the calling process.
2) To label the object type name with an operation and a link to the process image which implements the operation for processes running at the current directory node.
3) If <processimage> = <objectype>, the object type being registered corresponds to an executable file.

Parameters:

<objectype>: (input) This is the object type to enter into the directory.

<processimage>: (input) Process image name currently residing in the directory that is to be associated with <operation> on objects of type <objectype>.

<operation>: (input) A mnemonic for the operation privilege being granted to processes running at the current directory node. Can be 'S' (Send), 'R' (Receive), or 'SR' (Selectreceive).

<password1>: (input) A value to be used in determining whether the process is to be able to add an object type to its current subdirectory.

<password2>: (input) Password to enable Register of an object type connected to the processimage, or if Register is of a process image, i.e., <objectype> = <processimage>, a password to be supplied by a process in order to Register an object type connected to the process image.

<status>: (output) indication of the result of the register attempt
   0: Successful Register performed
   1: Error in calling sequence - illegal parameter
   2: Error - directory of <userid> is full.
   3: Error - process has no privilege to update directory.

CHANGEDIR (newuserdir,password,status)

Executed by: Any process.

Purpose: To change the directory of the running process.

Parameters:

<newuserdir>: (input) New directory node to be made the current node of the calling process. <newuserdir> must be among the current node's successors.

<status>: (output) Indication of success of change directory attempt
   0: Successful change directory performed.
   1: Error - <newuserdir> is not a directory object type in the current node of the calling process or the calling process does not have the correct privilege to make the change.

REMOVETYPE (objectype,password,status)

Executed by: Any process.

Purpose: To remove an object type from the subdirectory currently associated with the calling process.

Parameters:

<objectype>: The entry to be removed from the subdirectory currently associated with the calling process. <objectype> must be among the current node's succesors.

<status>: (output) Indication of success of removal attempt
0: Successful Removetype performed.
1: Error - <objectype> is not a directory object type in the current node of the calling process or the calling process does not have the correct privilege to make the change.

7. An Example Using Capabilities

In this section we present an example of using ports to implement a database system in order to illustrate use of the capability directory. We start by adopting the idea that a database system is itself a single abstract data type, where all legal database states constitute the value set of the type and all legal updates, including transactions, are the type's operations. One problem to be solved in implementing this view is to limit certain users (processes) to only particular operations on the database (the object of the type), while allowing other processes, such as the operation implementations themselves, more privileged capabilities.

Consider an example of a database on the National Football League maintained by a football fan using his or her personal system. One of the database operations, Hire-players, models the hiring of players by a team. Suppose that the part of the database affected by this operation consists of three database files, Players, Player-histories, and Team-stats. Suppose further that the user has written an interactive program which reads the hiring information from the user's terminal and builds a non-database file of new player records. The problem is how to allow the program to execute the Hire-players operation without allowing the program to gain access to the database files themselves.

To address this problem using the Gutenberg system, suppose that we have compiled the operation Hire-players and stored it as an executable file. Suppose also that we have built a sub-directory (Hire-players-dir in the figure below) which contains only those object types (database files) that the operation needs, namely Players, Player-histories, and Team-stats. The capability directory shown below can now be used to accomplish our goal.

```
                                       |
                                    USERDIR
                                       |
                          +------------+------------+
                          |                         |
                          |                         |
                       NFLdir                       |
                          |                         |
                 +--------+--------+                |
                 |                 |                |
                 |                 |                |
         Hire-players-dir     Hire-players          |
                 |               |      ^           |
                 |               | SR   |           |
                 |               +------+           |
                 |                                  |
       +------------------------+                   |
       |             |          |                   |
NFL-playerstats  NFL-teams  NFL-teamstats       Newplayers    Filemgr
       |             |          |                   |           ^
       |             |          |                   |   R,S     |
       |             |          |                   +-----------+
       |             |          |        R,S                    |
       |             |          +-------------------------------+
       |             |    R,S                                   |
       |             +------------------------------------------+
       |     R,S                                                |
       +--------------------------------------------------------+
```

Capability Directory

A program running with a capabilities link to USERDIR (no access to the database) could build a file of new players from information typed at a terminal (with the proper prompting and editing). If the program were written in PASCAL, this would be done with reads and writes in the standard way, but the run-time support package would execute a Createport to allocate a port for writing the Newplayers file and use Sends to write it. Terminal IO would be implemented as Sends and Receives on the standard terminal in and out ports. These would be created by the program initiation routine.

After writing the Newplayers file, the program needs to gain some kind of access to the database. To do this it changes its current directory pointer to point to NFLdir using the Changedir operation. This gives it the privilege of executing Hireplayers, but not to access any file of the database. Even so, changing the directory pointer to NFLdir must be a protected operation, unless any process is to be able to execute Hire-players. For this protection the Gutenberg system uses a password to enable a process to execute a Changedir.

After moving to NFLdir the program executes Hireplayers, by first creating a port for the purpose of executing a Selectreceive of the Hireplayers type, then executing a Selectreceive on the port, and sending a Receive port for the Newplayers file in the request details. This causes the creation of a Hire-players process with the capability of reading Newplayers on the passed port. The Hireplayers process now needs to gain the capability of reading and writing the database files and therefore changes its capabilities node to Hire-players-dir. This is a serious change of capabilities and should therefore probably be protected by something more than a password. Such protection is not currently designed into the Gutenberg, but is planned for in the future.

The following is a simplified skeleton of what the operating system interface calls would look like for both the user program and the Hireplayers operation.

```
User program
   Createport(Wr-newplayers-port,'Newplayers','Send',status)
   While more from terminal do

     begin

       Send(Termoutport,Termprompt,'Nowait',status)

       Receive(Terminport,Newplayerdata,'Wait',status)

       Send(Wr-newplayers-port,Newplayerdata,'Nowait',status)

     end

   Waitall(Wr-newplayers-port,status)

   (* Create a port for use in Hireplayers for getting Newplayers input. *)

   Createport(Rd-newplayers-port,'Newplayers','Receive',status)

   (* Change directory position so that Hireplayers is "visible". *)

   Changedir('NFLdir',NFLpassword,status)


   (* Execute Hire-players *)

   Createport(Exhireplport,'Hireplayers','Selectreceive',status)

   Selectreceive(Exhireplport,Hpmessbuffer,Rd-newplayers-port,'Wait',status)

   If status =0 then write('OK') else write('Nogo on Hireplayers.')


Hire-players

   (* Find out request details, in particular the port for reading input.*)

   Acceptrequest(...,Newplayers-port,...)

   (* Gain access to the database files.*)

   Changedir('Hire-players-dir',Hp-password,status)

   (* Open files for reading.*)

   Createport(Rd-players-port,'Players','Selectreceive',status)

   Createport(Rd-Teamstats-port,'Teamstats','Selectreceive',status)

   (* Open files for writing.*)

   Createport(Wr-Players-port,'Players','Send',status)

   Createport(Wr-Teamstats-port,'Teamstats','Send',status)

   Createport(Wr-Player-histories,'Player-histories','Send',status)
```

```
    .
    .
    .

(* Read Newplayers.*)

Receive(Newplayers-port,Player-rec,'Wait',status)
    .
    .
    .

(* Check player record and write it to the Players file.*)

Send(Wr-players-port,Player-rec,'Nowait',status)
```

8. Process Management

Process management is performed as a byproduct of the port management routines. Processes are created when a Send, Selectreceive, or Receive is executed and the server of the port is uninitiated. Createprocess, a procedure local to the port manager, is the routine called to create a process. It creates a process control block (PCB, see below) and returns a processid, essentially a PCB pointer.

Two system ports are established explicitly for process management. The currently running process is always queued on the runprocess port. The readyprocess port contains all processes in the ready state awaiting execution. In order to use ports for passing process ids from the port operation procedures to the dispatcher the port manager and the dispatcher together are identified as one "pseudo-process" merely for the purpose of owning the runprocess and readyprocess ports. This pseudo-process will be referred to as both the process manager and the dispatcher.

Processes are placed in the ready state by being sent (using the Send operation) to the dispatcher via the readyprocess port. Processes become ready to execute in four situations:

1. As part of Send, Selectreceive, or Receive on a port with an uninitiated server, the created process is sent to the dispatcher on the readyprocess port.
2. As part of a Send on a port which has a blocked receiver, the receiving process is unblocked and sent to the dispatcher on the readyprocess port.
3. As part of a Receive or Selectreceive on a port which has a blocked request accepter (Acceptrequest caller) or a sender with a full port, the blocked process is unblocked and sent to the dispatcher on the readyprocess port.
4. Upon receiving a timer interrupt at the end of its quantum for a process, the dispatcher sends itself the process on the readyprocess port.

The dispatcher places a process in the running state by sending it to a processor on the runprocess port. The details of context switching, processor status restoring, etc. are not dealt with in this report.

Blocked processes are recorded in the details of the ports. A process is blocked either by executing a Receive or a Selectreceive on an empty port, or by an Acceptrequest before a corresponding Send, Selectreceive, or Receive.

Below we detail the actions of each of the port operations.

Createport operation

CREATEPORT (portid,objectype,operation,status)

When Createport is executed, the user directory (located using the pointer in the PCB of the caller) is searched for the objectype passed in the parameter list. If the objectype is located in the directory, the capability for the object type (the label on the link to the process image) is checked to match the requested use of the port. Errors result if the objectype is not located or the capability for the requested operation does not exist. If the search and matching is successful a new port entry is added to the system port list. The new entry contains the objectype, directionality, process id and owneruse from the input parameter. The server's status is set to 'uninitiated', corresponding to an uninitiated server process, and the server process image name from the directory is placed in the new port entry. Then a new port id is generated corresponding to the port entry created, and is returned in the <portid> parameter to the caller.

Receive operations

RECEIVE (portid,buffer,waitnowait,status)

SELECTRECEIVE (portid,buffer,waitnowait,requestdetails,status)

When a receive operation is executed, there are a number of cases:

1. At least one object is queued on the port.

   Pop the object queue, setting the object in the buffer parameter.
   If the operation is a Selectreceive queue the request details
   Set the status and return.

2. The object queue of the port is empty:

   a) The port server is uninitiated:

      Use Createprocess to create a PCB;
      Set sender field of port entry to process id returned;
      Set the status of the server to 'initiated';
      Set 'Blocked receiver' field in port entry to 'T' for true;
      Send the server process id to the process manager
      using the readyprocess port, with the system parameter
      'processid' in the send call set to 'procmgrid';
      Call the dispatcher with 'Block' to remove the
      currently running process from the runprocess port and
      to insert a new process on the runprocess port;
      If the operation is Selectreceive, queue the requestdetails.

Return with status set appropriately.

b) The port server is initiated

If the server is blocked on a previous Acceptrequest,
then mark the server status in the port entry as
'not blocked' and send the server process id to the
process manager via the readyprocess port;
Mark the receiver as blocked in the port entry;
Call the dispatcher with 'Block' to remove the currently
running process from the runprocess port and
to insert a new process on the runprocess port;
If the operation is Selectreceive, queue the requestdetails.
Return with status set accordingly.

<u>Send</u> <u>operation</u>

SEND (portid,buffer,waitnowait,status)

When Send is executed, the object in the caller's buffer is queued on the port's object list, and there are 3 cases:

1. The port server is uninitiated:

   Use createprocess to create a PCB
   Set the receiver field of port entry to the process id
   Set the server status of the port entry to 'Initiated'
   Send the server process id to the process manager on the
   readyprocess port
   Return with status set appropriately

2. The receiver is not blocked:

   Return with status set appropriately

3. The receiver is blocked:

   Set the receiver status in the port entry to 'Unblocked'
   Send the process id of the receiver to the process manager
   on the readyprocess port
   Return with status set appropriately

<u>Acceptrequest</u> <u>operation</u>

```
ACCEPTREQUEST (objectype,portid,operation,directionality,
               requestdetails,waitnowait, status)
```

When Acceptrequest is executed, portid is set from the server port id in the calling process PCB and the object type from the port entry.

If the port id passed is a bidirectional port, the calling process must be the sending server of the port. There are two cases to be considered:

```
1. An unserved receive has been executed and thus
there are request details queued on the port.
   Pop the request details and return them in the
   requestdetails parameter.
   Set the status and return normally.

2. No unserved receive has been executed and thus
the request detail queue is empty. The calling
process is blocked on acceptrequest.
   Call the dispatcher to put a new process on the run port.
   Control returns normally.
```

If the port involved is a unidirectional port there are three cases to consider:

```
1. The calling process is a sending server and
there is a blocked receiver on the port.
   Return normally.

2. The calling process is a sending server and
there is no blocked receiver.
   If waitnowait is set to 'nowait', return with error.
   Otherwise, block the calling process by setting blkreq='Y'
   and call dispatch.

3. The calling process is a receiving server.
   Return normally regardless whether the port is
   empty or not.
```

The primary data structures of the port manager are the port entry and the process entry (PCB). Below is a schematic of the items in each of these structures.

Port entry items:

Portid:	Port Id
Objectype:	Object Type
Directionality:	directionality('B' or 'U')
Recid :	Receiver process id
Sendid :	Sender process id
Owneruse:	Owner use('S' or 'R')
Status:	status of server('I','U' or 'Hardware')
Blkrec:	Blocked receiver status ('Y' or 'N')
Blkreq:	Blocked request-waiter status ('Y' or 'N')
Objecthead:	Object list head
Objecttail:	Object list tail
Reqhead:	Request detail list head
Reqtail:	Request detail list tail
Image:	process image of server
Ownerlink:	link to next port owned by this port's owner
Serverlink:	link to next port served by this port's server

Process entry items (PCB)

Processid:	Process id
Image:	process image
Userid :	user id
Port-list-hdr:	link to first port owned by this process
Serve-list-hdr:	link to first port served by this process