The Prototype Gutenberg System
(version UM2.0)
[1]

Panayiotis K. Chrysanthis
Hanuma Kodavalla
Krithi Ramamritham
David Stemple

COINS Technical Report 88-25
March 1988

[1]This material is based upon work supported in part by the National Science Foundation under grants DCR-8403097, and DCR-8500332.

PREFACE

This document serves as a comprehensive description of the Gutenberg prototype kernel and is consistent with the current version of the system, *UM2.0*. It meets the needs of the Gutenberg system users, administrators and future developers. It also gives a quick introduction to the Gutenberg philosophy and its innovative ideas. Hopefully, these ideas, together with the functionality of the Gutenberg kernel, will attract users without getting them bogged down by the complexity of its lowest level.

This report is organized as follows: Chapter 1 is an introduction to the Gutenberg System. Chapter 2 describes the Gutenberg primitives and how they can be invoked from user programs. Chapter 3 presents the *Gutenberg Command Processor*, *gcp*. *Gcp* is a command interpreter or a shell that provides the interface between interactive users and the kernel. Chapter 4 discusses how to program the Gutenberg managers which constitute the basic units of any application structured within the Gutenberg environment. Chapter 5 talks about the implementation details of the kernel. Chapter 6 presents the administration aspects of the system. It also gives a brief description of the organization of the Gutenberg kernel development environment.

There are also three appendices: the first discusses a simple Gutenberg utility, the Gutenberg Error Manager; the second presents the code of the Error Manager, and the third illustrates the script used to introduce the manager into the Gutenberg environment.

ACKNOWLEDGEMENTS

The Gutenberg project was started at the Computer and Information Science Department of University of Massachusetts at Amherst in 1983. Steve Vinter worked on the design of the Gutenberg Kernel leading to a Ph.D. thesis in September of 1985. The design and implementation of the first Gutenberg prototype kernel was completed a year later by Panayiotis Chrysanthis that being his M.S. project. David Briggs and Hanuma Kodavalla had contributed with ideas and support during the first and the second phases of the project, respectively. V.R Govindarajan implemented the checkpointing module of the Kernel. Later, Hanuma Kodavalla, as part of his M.S. project, with the help of Panayiotis Chrysanthis, expanded and optimized the prototype kernel bringing it to the current version.

Contents

1 Gutenberg System **1-1**

1.1 Introduction . . . 1-1
1.2 Gutenberg Environment . . . 1-1
1.3 Gutenberg Kernel . . . 1-3
1.3.1 Ports . . . 1-3
1.3.2 Interconnection Schema . . . 1-5
1.3.3 Transient Capabilities . . . 1-10
1.6 Gutenberg Publications . . . 1-14

2 Gutenberg Primitives **2-1**

2.1 Introduction . . . 2-1
2.2 Errors . . . 2-11
2.3 Acptrqst . . . 2-13
2.4 Associate . . . 2-14
2.5 Changedir . . . 2-15
2.6 Compare . . . 2-16
2.7 Copy . . . 2-18
2.8 Create . . . 2-19
2.9 Destroy . . . 2-23
2.10 Drop . . . 2-25
2.11 Examine . . . 2-26
2.12 Getdetails . . . 2-27
2.13 Hold . . . 2-29
2.14 Home . . . 2-30
2.15 Merge . . . 2-31
2.16 Modify . . . 2-33
2.17 Queryport . . . 2-35
2.18 Receive . . . 2-36
2.19 Refuse . . . 2-38
2.20 Register . . . 2-39
2.21 Reject . . . 2-40
2.22 Remove . . . 2-41
2.23 Revoke . . . 2-42
2.24 Send . . . 2-43
2.25 Sendrecv . . . 2-45
2.26 View . . . 2-47

3 Gutenberg Command Processor **3-1**

3.1 Introduction . . . 3-1
3.2 GCP Commands . . . 3-3
3.2 Associate . . . 3-3
3.3 Changedir . . . 3-3
3.4 Compare . . . 3-4
3.5 Copy . . . 3-4
3.6 Create . . . 3-5
3.7 Destroy . . . 3-5
3.8 Drop . . . 3-6

3.9 Hold . . . 3-6
3.10 Home . . . 3-7
3.11 Logout . . . 3-7
3.12 Merge . . . 3-7
3.13 Modify . . . 3-8
3.14 Register . . . 3-8
3.15 Remove . . . 3-9
3.16 Script . . . 3-10
3.17 View . . . 3-11
3.3 GCP Library . . . 3-12

4 Programming Gutenberg Managers **4-1**
4.1 Introduction . . . 4-1
4.2 Manager Definitions . . . 4-1
4.3 Manager Program Structure . . . 4-2

5 Implementation Details **5-1**
5.1 Introduction . . . 5-1
5.2 System Organization . . . 5-1
5.3 Data Structures . . . 5-3
5.3.1 Capability Directory . . . 5-3
5.3.2 Process Control Block . . . 5-9
5.3.3 Channel Control Block . . . 5-11
5.3.4 Mailbox . . . 5-15
5.4 System Modules . . . 5-17
5.4.1 Kernel Control Manager . . . 5-17
5.4.2 Capability Directory Manager . . . 5-20
5.4.3 Process Manager . . . 5-23
5.4.4 Port Manager . . . 5-24
5.4.5 Run-Time Support (rts) . . . 5-27
5.4.6 Input/Output . . . 5-28
5.5 Life-Cycle of a Process . . . 5-28

6 System Administration **6-1**
6.1 Introduction . . . 6-1
6.2 Gutenberg System Directory Tree Structure . . . 6-1
6.2.1 Minimal Directory Tree Structure . . . 6-1
6.2.2 Development Directory Tree Structure . . . 6-1
6.2.3 Auxiliary Directories . . . 6-3
6.3 Interconnection Schema Structure . . . 6-3
6.4 Booting Process . . . 6-5
6.5 Interconnection Schema Construction . . . 6-6
6.6 Monitor Facility . . . 6-6

Appendix
A Design of a Manager . . . A-1
B Source Code of a Manager . . . B-1
C Script to create a Manager . . . C-1

1 Gutenberg System

1.1 Introduction

Gutenberg, an object-oriented operating system kernel, is designed to facilitate the design and structuring of distributed systems. The main goal of the Gutenberg system is to build an environment that supports the development of manageable and understandable systems.

In large distributed systems lack of a clean, well-structured paradigm for processes and their interactions leads to more complexity than is required by the nature of the distribution alone. Gutenberg attempts to solve this problem in an efficient manner by taking a unique approach to interprocess communication. The salient features of the Gutenberg paradigm are: use of ***port-based communication***, whereby processes can communicate only by means of ***ports***, queue-like objects which are managed by the kernel; ***non-uniform object orientation***, whereby only interprocess communication, but not module interconnection within a process, is structured and controlled by means of capabilities; and decentralized access authorization using ports and the ***Interconnection Schema*** [1]. The Interconnection Schema is the driving force of the whole system. It expresses all the potential process interconnections and provides an abstract view of the functional building blocks of the executing systems within the Gutenberg environment. The Interconnection Schema is a distributed persistent object, maintained by the kernel. Processes can manipulate the Interconnection Schema and ports using kernel-implemented operations called ***primitives***.

This chapter serves as an introduction to the Gutenberg system. The next section describes the Gutenberg environment and discusses the principles behind its design. The rest of the sections concentrate on the Gutenberg Kernel, presenting its primitives (system calls) and the structure of its objects. It concludes with a list of papers on which this chapter is based.

1.2 Gutenberg Environment

In Gutenberg, a distributed system is a group of ***objects*** that execute asynchronously and concurrently, interacting cooperatively to perform a task. Objects are implemented using processes (independently schedulable units of computation) which are hidden from each others' views. Processes are also referred to as the object ***managers***. Managers synchronize operations on their objects and are the only subjects able to directly manipulate the objects.

Gutenberg objects can communicate only through explicit message exchange over communication channels called ***ports***. Objects do not share address spaces. Since objects are instances of abstract data types, interprocess communication in Gutenberg is always in terms of requests for abstract data type operations. While Gutenberg enforces an object-oriented view on all interprocess communication, it does not enforce this view upon the programs running in a single process. The organization of intraprocess communication depends on the programming language used to build the process program and can be object-oriented or not.

Ports between objects are established based on the need to provide or request a service. That is, Gutenberg has adopted the client/server model for basic interprocess communication in which the user of a port, called the client, sends a request for an operation to the port server, the object's

[1] In early papers, the Interconnection Schema was called *Capability Directory*.

manager, which then performs the operation and may send back a reply. Objects can simultaneously be clients and servers.

A port is established using *functional addressing*. A client creates a port by naming the service (the operation) it would like to request using the port rather than identifying the server object. The advantage of this strategy is that it supports service transparency, allowing for dynamic object re-implementation and/or relocation, an important property for distributed systems. The client object does not have to know the identity of the server object, or whether that object executes on a local or remote machine. The server object does not even have to be in existence prior to the creation of the port. Manager instantiation and destruction in Gutenberg is a side-effect of port operations. There are no primitives to instantiate or destroy object managers explicitly. Furthermore, object manager interconnections can be dynamically changed by transferring ports over other ports. In order to restrict the use of ports to the functionality for which they have been created, a port is typed with respect to its directionality and message contents.

The kernel enforces a port-based access control to objects. An object A can create a port for requesting an operation on an object B only if it has the *capability* to execute that operation. After port creation, the only check that needs to be made when the object A requests access to the remote object B via this port is whether that object A has the capability to access that port. This required check is done at the node in which the object A executes and so it is a local check.

The capabilities for creating ports are stored in the *Interconnection Schema*, a persistent, distributed object managed by the kernel. Thus, the Interconnection Schema expresses all the potential object interconnections achievable by programs running under the kernel's control. This means that the Interconnection Schema represents the organization of applications runnable under the Gutenberg kernel at any given time. Besides enforcing interconnection structure, the Interconnection Schema also supplies the kernel the information needed to locate, and if necessary instantiate, the server of a port. It also contains the definitions of object managers, that is, components that provide access to the code that implements the operations of objects, and the rules for activating the object managers. New managers are introduced into the system by storing their definitions in the Interconnection Schema (for more details on how to program a Gutenberg manager, see chapter 4).

The Interconnection Schema is not the only repository for capabilities in Gutenberg. There are also *transient* capabilities which persist only as long as the owner object manager is active, and these are stored in lists associated with manager processes. At any time, each object manager in the system is associated with a segment of the Interconnection Schema, designated as its *active directory*, and with its transient capabilities stored in its *capability list*, abbreviated *c-list*. Each object manager is associated with a single c-list which cannot be shared. Objects managed by the same manager can share their manager's c-list. Gutenberg supports dynamic access control by allowing objects to traverse the Interconnection Schema acquiring new capabilities, or by transferring capabilities over ports.

1.3 Gutenberg Kernel

Processes view the kernel as an abstract data type manager of kernel objects. The kernel primitives are the corresponding abstract data type operations. Processes, however, are not required to create a port for invoking kernel primitives. Processes can directly invoke a kernel primitive which is trapped by the kernel. ***Ports***, ***Interconnection Schema*** and ***capabilities*** are the kernel objects, that processes can manipulate and share.

1.3.1 Ports

A Port is a communication channel between a pair of processes. At any time, only two processes have the capability to access a port for either placing messages or removing messages. A port behaves as a queue of messages awaiting delivery. Communication through a port can be synchronous or asynchronous. The representation of the port and the details of message transmission and reception are all hidden from the communicating processes.

Processes access ports by invoking kernel primitives. Ports are strongly typed with respect to their directionality, the format of the message, and their use. The directionality of the port could be *Send*, *Receive* or *SendReceive*. Send and Receive ports are unidirectional. SendReceive ports are bidirectional, allowing the port's client to send a message and receive a response from the port's server.

Figure 1.1 shows the port primitives that clients and servers may use for each port type. A client invokes the CREATE-PORT primitive to create a port of a specific type for requesting an operation on an object. The kernel attaches the newly created port to the appropriate server which is required to invoke the ACCEPT-REQUEST primitive to obtain access to the port. A client requests an operation over the port by invoking the appropriate primitive supported by the type of the port. The server can choose either to service the request by invoking the appropriate port primitives, or to refuse service by invoking the REFUSE primitive. A server can use the EXAMINE primitive to decide which port to service next and which requests to refuse. A client can abort the last pending request on a port by invoking the REVOKE primitive or even destroy the port with DESTROY. The server of a port can permanently refuse to service a port by invoking REJECT which results in the destruction of the port.

The bidirectional SENDRECEIVE primitive and its receiving counterpart GETDETAILS are provided, in addition to the basic SEND and RECEIVE primitives, in order to allow for remote procedure call semantics to be implemented by a single primitive. A single primitive is better for both performance and software engineering reasons (less prone to errors than a pair of primitives). SENDRECEIVE is more robust and flexible than a remote procedure call because it supports both synchronous and asynchronous modes.

The owner of the client-end port capability can *destroy* the port; whereas the owner of the server-end port capability can *reject* the port. The creator of a port becomes the initial owner of the client-end port capability. The first server of a port becomes the owner of the server-end port

Port Type	Client Primitives	Server Primitives
any	CREATE-PORT DESTROY REVOKE	ACCEPT-REQUEST REJECT REFUSE
Send	SEND	RECEIVE EXAMINE
Receive SendReceive	RECEIVE SENDRECEIVE	SEND GETDETAILS SEND EXAMINE

Figure 1.1: Port primitives used by clients and servers for each port type

capability. As part of the sharing mechanism supported by the Gutenberg system, a process may transfer part of its capabilities, including port capabilities, to another process through a port.

Here is a short summary of the functionality of port primitives. Their interface and details of their invocation are described in the following chapter on *Gutenberg Primitives*.

CREATE-PORT (a client primitive) creates a port of a specific type for requesting a specific operation.

DESTROY-PORT (a client primitive) destroys a port. The caller must be the owner of the port.

SEND puts a message on a port. The system has two kinds of SEND primitives: acknowledge-SEND and no-acknowledge-SEND. If the SEND is an acknowledge-SEND, the sending process is informed when its correspondent over the port receives the message. The sender can choose to block until the receipt of the acknowledgement.

RECEIVE requests the next message from the port. If there is no message on the port, the caller may elect to either block, or execute concurrently with the servicing of the request.

SENDRECEIVE (a client primitive) puts information, termed *request details*, on a port for the server to use in satisfying the request. When the server responds to the request by executing a SEND, the server's reply is returned to the client as in RECEIVE. The caller may block until the server replies, or execute concurrently with the servicing of the request.

ACCEPT-REQUEST (a server primitive) is used to obtain access to newly created ports.

QUERY-PORTS (a server primitive) is used to query a set of existing ports to see if new messages have arrived.

GETDETAILS (a server primitive) gets request details from a port. The caller (the port's server) may block if there is no pending SENDRECEIVE, and thus no request details, on the port, or it may execute concurrently with the satisfaction of its request.

EXAMINE (a server primitive) examines messages on the port without removing them.

REFUSE (a server primitive) rejects a client's request for service and notifies the client by setting a status.

REJECT (server primitive) rejects service to a client by destroying the port. The caller must be the original server of the port.

REVOKE (a client primitive) revokes the last request over a Send, Receive or SendReceive port up to the receipt of the request.

1.3.2 Interconnection Schema

The Interconnection Schema is a repository of type descriptors and capabilities. It is a logically unified but physically distributed structure maintained by the kernel. It contains the information needed by the kernel to control the creation and use of ports. Specifically, it contains information about the services and their servers (acts as a name server), process creation protocols, circumstances under which a process can request a given service (process environment) and dynamics of access control (capabilities). The Interconnection Schema is similar to the UNIX file directory in the sense that it provides uniform treatment of files, devices and interprocess communication.

The Interconnection Schema is a *stable* structure in that its existence does not depend on the existence of any process. It is also *shared* since more than one process may concurrently access the same segment of the schema.

Interconnection Schema Components

The Interconnection Schema is structured as a directed graph. The nodes of the graph correspond to the Interconnection Schema components, called *interconnection schema nodes*, abbreviated as *is-nodes*. The edges of the graph correspond to capabilities. Is-nodes contain various information along with capabilities. Is-nodes are identified by user-specified names stored in the capabilities that point to them. The same is-node may be linked to several is-nodes under possibly different user-specified names. All capabilities that point to an is-node have equal status. Is-nodes are unique and are *not* contained within other is-nodes. An is-node exists independently of any other is-node and disappears along with the last capability link to it, unless it is explicitly destroyed.

The Interconnection Schema may contain two kinds of is-nodes: *directories* and *manager definitions*. Directories are the structural components of the Interconnection Schema whereas manager definitions are the operational components of the Interconnection Schema.

A directory is a list of capabilities. It is an organizational unit of the Interconnection Schema, similar to a file directory in a file system. At any time, every process in the system is associated with a single directory in the Interconnection Schema designated as its *active directory*. The active directory of a process defines its execution environment. It is the set of capabilities from the Interconnection Schema that a process may use. A process may dynamically switch from one

execution environment to another by changing to a new active directory or changing the contents of its current active directory, if it has the capability to do so.

Manager definitions are *Abstract Data Type* definitions. All processes in the system are instantiated from manager definitions as a result of port operations. The kernel creates a process to serve a newly created port if another appropriate server does not already exist. However, for better resource management, a process is allocated resources and becomes active only after the first operation on one of its ports.

Each manager contains the following information which is necessary for instantiating a manager process:

> **Initial active directory**: a directory capability for the directory that becomes the active directory of all object managers instantiated from the manager definition.
>
> **Process image**: a capability representing the file containing the object code to be executed when the process is initiated.
>
> **lifetype**[2]: indicates whether a new manager process is created or an existing manager process is connected to service a new port.
>
> **Operation descriptors**: This is the set of operations defined within the manager. Each operation description contains a ***generic operation name*** (the name of the operation known by the manager), and the type of port through which a user requests this operation.

Capabilities in Gutenberg consist of three parts: a specific kernel primitive (the *primary* primitive), a list of parameters for the primitive, and a list of primitives that can be used to manipulate the capability itself, which are called *capcaps*, for capabilities on a capability.

A capability permits a process which possesses it to invoke the primary primitive it contains. The invocation of the primary primitive allowed by a capability is known as *exercising* the capability.

The capcaps determine how the capability may be modified and used. Capcaps include the privilege to *transfer* (to another process), *copy*, *register* (make stable), *hold* (make transient), *merge* (with other mergeable capabilities), *view*, *modify*, and *associate* the capability. Each capcap may be active, in which case the corresponding kernel primitive may be invoked for the capability, or inactive, in which case the corresponding kernel primitive cannot be invoked on the capability. Not every capcap makes sense for each type of capability. When we discuss the specific capabilities, we point out the capcaps that are applicable.

The parameter list may include names of is-nodes as well as other capabilities (most notably, the *cooperation class* capability which is discussed later). The parameter list is initialized at the creation time of the capability by the kernel. Subsequently, some parameters can be modified by processes if they have the capability to do so.

---

[2]known also as *instatiation protocol*

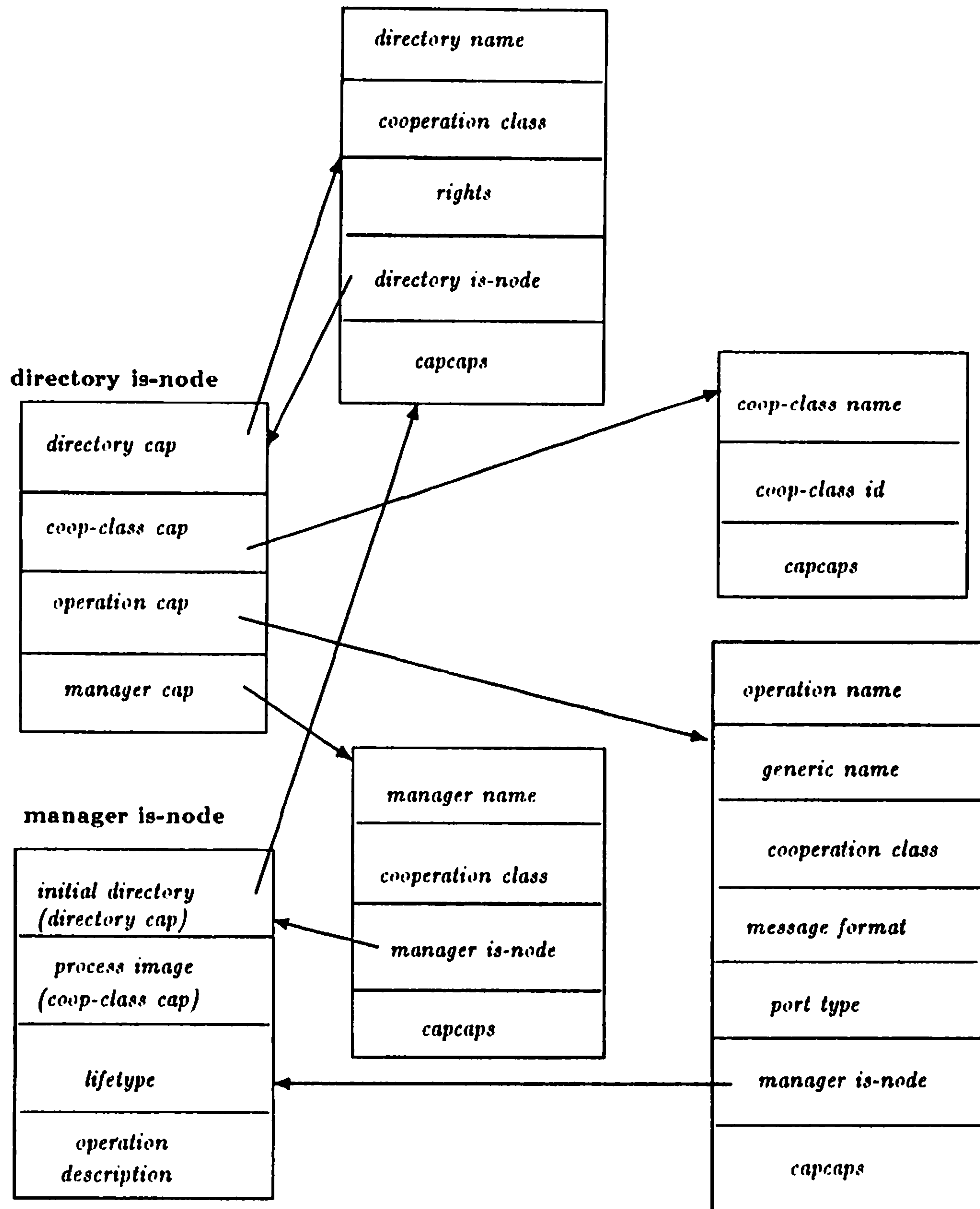


Figure 1.2: Components of the Interconnection Schema

Types of Capabilities

There are four different types of capabilities that may be stored in the Interconnection Schema: *directory, manager definition, operation*, and *cooperation class capabilities* (figure 1.2).

A directory capability points to a directory is-node. The primary kernel primitive associated with the directory capability is **CHANGE-DIRECTORY**. The CHANGE-DIRECTORY primitive is used by a process to change its active directory to the directory named in the directory capability.

An important attribute of a directory capability is a set of *directory rights*. When a directory capability is used to make a directory active, the directory rights override (mask) the capcaps of each individual capability in the directory. This further restricts the use of the capabilities registered in the directory.

Here is a complete list of the attributes of a directory capability:

Directory name: user-specified name of the directory is-node which becomes the active directory for the process when the CHANGE-DIRECTORY primitive is invoked. This name also serves to identify the directory capability.

Cooperation class: user-specified name of the cooperation class that this directory belongs to. This is used to further restrict which managers may make this directory their active directory. When the CHANGE-DIRECTORY primitive is invoked, the caller must possess a corresponding cooperation class capability or else the primitive is illegal.

Directory rights: restrict how is-nodes and capabilities contained in the directory may be used; the rights are: TRANSFER, COPY, REGISTER, REMOVE, HOLD, MERGE, VIEW-CAP, VIEW-NODE, MODIFY, DESTROY-MANAGER-NODE, DESTROY-DIR-NODE, ASSOCIATE, CHANGE-DIRECTORY, CREATE-OPERATION, and CREATE-PORT.

Directory id: a pointer to the directory is-node corresponding to this capability.

Capcaps of the directory capability: COPY, TRANSFER, REGISTER, REMOVE, HOLD, DESTROY-NODE, MERGE, ASSOCIATE, MODIFY-CAP, MODIFY-CAPCAP, VIEW-NODE and VIEW-CAP.

A manager definition capability points to a manager definition is-node. The primary primitive in the manager definition capability is **CREATE-OPERATION**, which is used to create operation capabilities, linked to the manager definition named in the capability. A manager capability has the following attributes:

Manager definition name: user-specified name used to identify the manager definition is-node to which the created operation capabilities are linked. This name also serves to identify the manager definition capability.

Cooperation class: user-specified name of the cooperation class that this manager belongs to. This is used to further restrict which managers may create operation capabilities linked to the manager definition. When the CREATE-OPERATION primitive

is invoked, the caller must possess a corresponding cooperation class capability or else the primitive is illegal.

Manager id: a pointer to the manager definition is-node corresponding to this capability.

Capcaps of the manager definition capability: COPY, TRANSFER, HOLD, REGISTER, REMOVE, DESTROY-NODE, MERGE, MODIFY-CAP, MODIFY-CAPCAP, ASSOCIATE, VIEW-CAP, and VIEW-NODE.

An operation capability represents the privilege to create a port for use in requesting a particular operation on a given object. The primary kernel primitive of the operation capability is **CREATE-PORT**. An operation capability has the following attributes:

Operation name: user-specified name of the operation. This name becomes the operation requested via ports created from this capability. It can be the same as or different from the generic operation name below. This name also serves to identify the operation capability.

Generic operation name: the name of the corresponding operation defined in the manager definition is-node.

Cooperation class: restricts how the port will be connected to a manager process.

Port type: specifies the type of the port needed to request the operation. The port type can either be Send, Receive, or SendReceive.

Message format:

specifies the arguments that may be passed over the port as part of the message, and the request-details in case of SendReceive port type. The argument types could be any capability type or just non-privileged data.

Manager id: a pointer to the manager to which this operation capability is linked.

Capcaps of operation capability: COPY, TRANSFER, REGISTER, REMOVE, HOLD, MERGE, ASSOCIATE, MODIFY-CAP, MODIFY-CAPCAP, and VIEW-CAP.

The fourth type of capability, the cooperation class capability, is a unique feature of Gutenberg. Unlike the other three types of capability the cooperation class capability is not a traditional capability in the sense that is associated with an object which it protects. It represents the privilege of an object to participate in a cooperative activity and thus, it can be associated with any object, client or server.

The cooperation class capability is a generic capability and as such it has no associated primary primitive. Instead, the cooperation class capability may be associated with any other capability type with whose primary primitive the cooperation class capability will be associated. The semantics of a cooperation class capability vary with the type of capability it is associated with.

Generally, the processes possessing a cooperation class become members of a cooperative group. In this case, the semantics of the cooperation class capability are that of an identification token for the cooperating group.

When a cooperation class capability is associated with either a directory or a manager definition capability, it restricts the invocation of the corresponding primary primitive to the processes which possess a corresponding cooperation class capability. In this way, directories and manager definitions participate in a cooperative activity represented by the capability. The semantics of the cooperation class capability are similar to that of a password and complement the role of capcaps and rights.

When a cooperation class capability is associated with an operation capability, the user need not possess the cooperation capability to invoke create port primitive (in contrast to the primary primitives of directory and manager capabilities). However, it restricts the way the created port is attached to the server. When the server accepts the new port, it gets the used cooperation capability into its possession.

Here, cooperation class capability can be used to identify either an instance of a manager (see manager's lifetype or instantiation protocols in chapter 4), or a particular instance of an object within a manager (a file for example). It can also be used as a synchronization token. In summary, cooperation class capabilities facilitate the communication among objects/managers that have no means of addressing each other explicitly.

New cooperation class capabilities can be created, on request, by the kernel and have the following attributes:

Class name: user-specified name used to identify the cooperation class in the current protection domain.

Class id: system wide unique identification of cooperation class.

Capcaps of the cooperation class capability: COPY, TRANSFER, HOLD, REGISTER, REMOVE, MERGE, VIEW-CAP, MODIFY-CAP, and MODIFY-CAPCAP.

Figure 1.3 shows a segment of an Interconnection Schema.

1.3.3 Transient Capabilities

Two features distinguish the transient capabilities from the stable capabilities: they are owned by a single process, and therefore cannot be shared, and their existence is dependent on the existence of the process that owns them. All the transient capabilities a process owns exist only for the duration of the existence of the process and are destroyed when the process terminates. The transient capabilities possessed by a process, are stored in the process' *capability list*, or *c-list*. A process, in addition to the capabilities in its active directory, may exercise the capabilities in its c-list. In particular, any search for a capability starts from a process' c-list and if it fails, it proceeds to the process active directory.

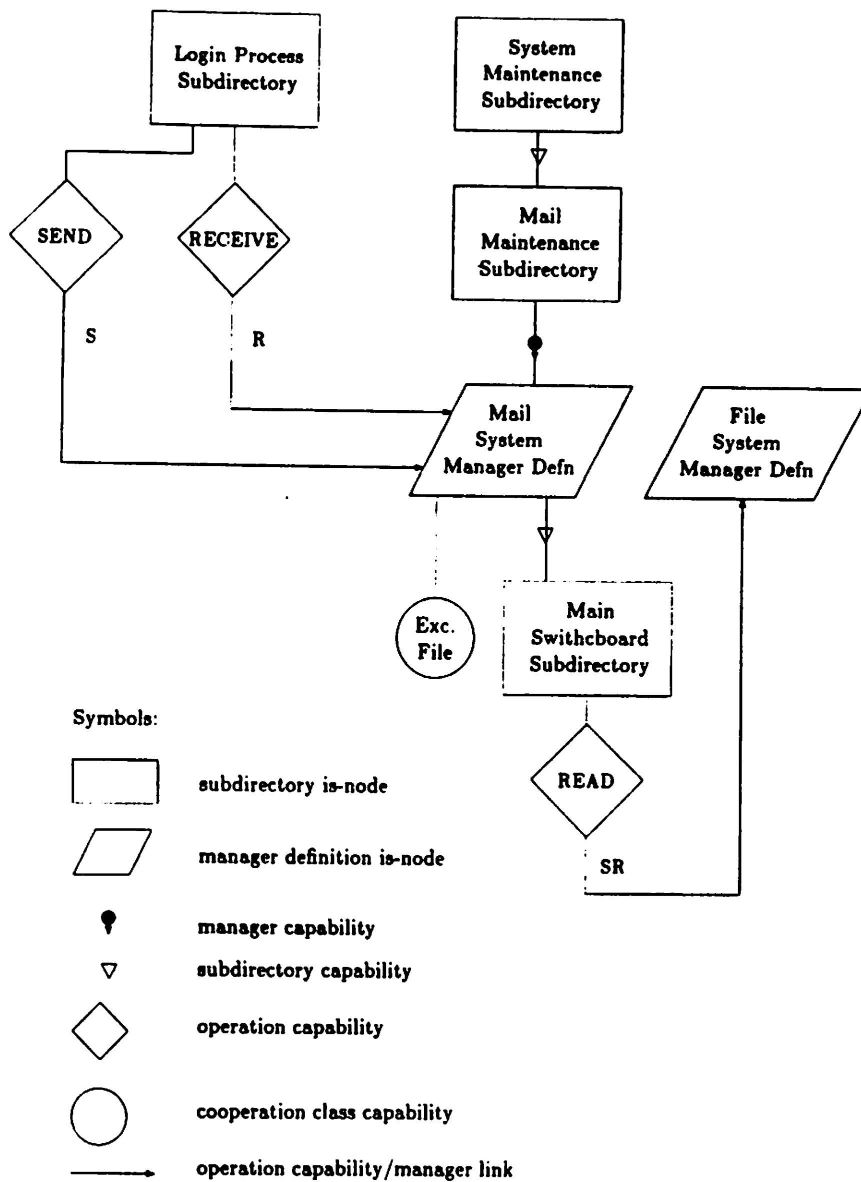


Figure 1.3: Example of Interconnection Schema Segment

This is a sample Interconnection Schema of a basic mail system. The Mail Maintenance Directory which contains the Mail System Manager Definition is created by the creator of the system and stored in the System Maintenance Directory. The two operations defined on the Mail System, namely the Mail and Read, are contained in the login process directory. Each user gets capabilities for these operations during the login procedure. In the initial active directory of the Mail System, operations on the File Manager exist to permit access to files.

A c-list is similar to a directory in the Interconnection Schema, but it is not part of the Interconnection Schema. A c-list is the local or working directory of a process. When a process changes to a new active directory, the capabilities in its c-list are neither affected nor constrained. The directory rights have no effect on the capabilities in the c-list. Transient capabilities contribute to the flexible use of capabilities in Gutenberg and as they are stored within the kernel space, they don't compromise the security of the system. Although transient capabilities cannot be shared, they are considered as kernel objects, and may only be manipulated through kernel primitives.

A transient capability comes into existence when a process copies or moves a capability from its active directory, into its c-list (using a primitive called HOLD); when it receives the capability from another process via a port; when it creates a capability by invoking the proper create primitive; or when it creates a port. In the last instance, two port capabilities are generated to access the created port, one for the client of the port and the other for the server of the port. A process copies or moves a capability in its active directory into its c-list in order not to lose the capability when it changes its active directory to another directory.

The port capabilities are *exclusive* and as such, are inherently transient, cannot be shared (become stable), or copied. However, either the client or the server process may transfer its port capability to another process. The transfer may be either temporary (the semantics of a lend with the ownership retained) with the SENDRECEIVE primitive or permanent with the SEND primitive. If the SEND primitive is used in the transfer, the ownership of the corresponding port-end is transferred along with the port capability.

The only capcaps associated with a port capability are ***modify*** and ***transfer***. The transfer capcap in the client's port capability is set to the value of the corresponding transfer capcap in the operation capability used. If the used operation capability is stable, then the transfer right of the client's active directory is also taken into consideration. The transfer capcap in the port capability of a server is always enabled. A server process may transfer a port that it serves, to another server as long as the port functionality is preserved. This is essential for supporting the implementation of load balancing algorithms and realizing a hierarchy of object managers. A process may reset the transfer capcap of the port capability when it passes the capability to another process, to prevent further transferring.

The kernel provides eleven generic primitives for manipulating the capabilities both within a c-list and an active directory. Recall that a certain capcaps and rights correspond to each primitive, and must be active in the capability on which the primitive is invoked, and must be allowed by the rights of the active directory of the invoking process.

Here is a brief description of generic primitives. More details concerning these primitives may be found in the chapter on *Gutenberg Primitives* together with the port primitives.

CREATE primitives create a capability. CREATE-OPERATION, and CREATE-PORT are instances of this generic primitive. A CREATE always creates a transient capability which may then be registered or transferred with all or part of its privileges retained. Creating an operation capability requires a manager capability, and creating a port capability requires an operation

capability. Other CREATE primitives require no privilege.

REGISTER makes a transient capability, stable. The purpose of these primitives is two-fold: to allow a process to store a capability for future reference in another session; and, to allow a process to share a capability with other processes which are not currently instantiated, but share access to a directory. This primitive can be either *constructive* in that it makes a copy of the transient capability or *destructive* in that it removes the transient capability from the caller's c-list.

REMOVE deletes a stable capability from an active directory.

DROP deletes a transient capability from a c-list.

HOLD makes a stable capability, transient. The purpose of these primitives is to allow a process to retain a capability from its active directory when it changes its active directory to another directory. This primitive can be either *constructive* in that it makes a copy of the stable capability or *destructive* in that it removes the stable capability from the caller's active directory.

VIEW These primitives bring a copy of a transient or stable capability, or an is-node into the address space of a process. Partial views are also facilitated. The purpose of these primitives is to allow a process to examine capabilities it possesses, and, if desired, use this information to modify or create new capabilities.

MODIFY These primitives allow a process to modify an existing transient or stable capability.

MERGE finds the union of the capcaps and the union of rights of two compatible capabilities and assigns them to the corresponding fields of the first capability. Two capabilities are compatible if they are of the same type, linked to the same is-node and belong to the same cooperation class.

COMPARE allows a process to compare two capabilities of the same type. It supports three comparison types: *compatibility check* by checking if the capabilities are linked to the same is-node and belong to the same cooperation class; *cooperation check* by checking whether the capabilities belong to the same cooperation class; and *object check* by checking if the capabilities are linked to the same is-node.

COPY creates a copy of a capability.

ASSOCIATE (de)associates a directory, manager or operation capability with a cooperation class capability.

As has previously been discussed, the manager definition and directory capabilities are slightly different from other capabilities. These capabilities contain pointers to manager definition and directory is-nodes, respectively. When one of these capabilities is created with the CREATE primitive,

the is-node is created as well. These is-nodes exist in the system until either they are explicitly destroyed by using the DESTROY primitive, or all of the capabilities that point to them are deleted using the REMOVE or DROP primitives.

All capabilities in the system are uniquely specified by the user-defined local name of the object they protect, the user-defined local name of the cooperation class to which they belong and their type. Two capabilities with the same specifications cannot exist in a directory or a c-list. For this reason all primitives that add a capability in a directory or c-list will fail if another capability with the same specifications already exists in the directory or c-list.

In summary, the kernel uses the c-list and the active directory of a process to check whether it has a legitimate privilege for executing a primitive it has requested. This primitive may be used either to manipulate a kernel object or to request an operation on an object managed by another process.

1.4 Gutenberg Publications

This section lists the published papers on Gutenberg. These documents describe in more details various aspects of the Gutenberg kernel and present the design decisions concerning the nature of communication and protection in the system. A complete descriptions of design decisions of the entire system can be found in Steve Vinter's Ph.D. Thesis:

Vinter, S. T., 'A Protection Oriented Distributed Kernel,' Ph.D. Thesis, University of Massachusetts, August 1985.

Here is the list of published papers:

Vinter, S. T., Ramamritham, K., Stemple, D., 'Protecting Objects through the use of Ports,' *Proceeding of the Phoenix Conference on Computers and Communication,* March 1983.

Stemple, D., Ramamritham, K., Vinter, S., Sheard, T., 'Operating System Support for Abstract Database Types,' *Proceedings of the 2nd International Conference on Databases*, September, 1983.

Ramamritham, K., Vinter, S. T., Stemple, D., 'Primitives for Accessing Protected Objects,' *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems,* October 1983.

Ramamritham, Stemple, D., Vinter, S. T., 'Decentralized Access Control in a Distributed System,' *Proceedings of the 5th International conference on Distributed Computing Systems*, May 1985.

Ramamritham, K., Briggs, D., Stemple, D., Vinter, S. T., 'Privilege Transfer and Revocation in a Port-Based System,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, May 1986.

Vinter, S. T., Ramamritham, K., Stemple, D., 'Recoverable Communicating Actions,' *Proceedings of the fifth International Conference on Distributed Computing Systems,* May 1986.

Chrysanthis, P.K., Ramamritham, K., Stemple, D.W., Vinter, S.T., 'The Gutenberg Operating System Kernel,' *Proceedings of the First ACM/IEEE Fall Join Computer Conference*, November, 1986.

Stemple, D., Vinter, S., Ramamritham, K., 'Functional Addressing in Gutenberg: Interprocess Communication Without Process Identifiers,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 11, December 1986.

2 Gutenberg Primitives

Introduction:

This chapter describes all of the Gutenberg primitives which implement the interface between the kernel and the user programs. This interface is referred to as *run-time support (rts)*. The *setenv* and *clnenv* procedures which setup and clean up the environment in which a manager runs, are the only two calls that are part of the run-time support which are not described in this chapter since these are not actual Gutenberg primitives. The run-time support is contained in the library called *rts.a*. The run-time-support routines are included in the *sdl.a* (software development library) as well.

All capabilities in the system are uniquely specified by the user-defined local name of the object they protect, the user-defined local name of the cooperation class to which they belong and their type. Two capabilities with the same specifications cannot exist in a directory or a c-list. For this reason all primitives that add a capability in a directory/c-list will fail if another capability with the same specifications already exists in the directory/c-list. Two capabilities are *compatible* if they represent the same primary privilege, that is they point to the same node in the Interconnection Schema, belong to the same cooperation class and are of the same type, but they have different local names for the is-node and/or cooperation class.

All primitives are invoked with a *status* parameter which contains the outcome of the call. All return error codes are *negative* integers while *zero* is returned on successful calls. All the constant definitions of the return errors are contained in the *"gtnerr.h"*. *"Gtnerr.h"* need not be explicitly included since it is contained within the *"shell.h"* which is required to be included in any manager as stated in chapter 4 on *Programming Gutenberg Managers*. The *"shell.h"* is a collection of include files and global variables which are used by the interface between the kernel and the managers. Among these files are the *"prmstr.h"* which declares the parameter structures used in a number of primitives; and, *"knldef.h"* which defines the constant values. The contents of the file *"gtnerr.h" is listed in the next section* **ERROR**(rts). *The files "prmstr.h", and "knldef.h"* are listed below.

As discussed in chapter 1 the basic Gutenberg interprocess communication is based on the *client/server model* where ports are used for requesting an operation on an object from its manager. Thus, the terms *client* and *user* are used interchangeably throughout this chapter. The terms *message* and *request* are also used interchangeably since a request is requested by sending a message on the appropriate port.

Some primitives as, for example, CREATE(rts), are implemented by a number of non-overlapping calls. The term *directive* is used to refer to such calls.

The typing conventions of this manual are the following:

> `Typewriter` characters are used to illustrate words that have been defined by the system. These are commands, types and constants.

italics are used in general to emphasize sections of the text. In particular, the parameters of the primitives are printed in italics.

Boldface is used to mark the names of the generic primitives which correspond to the indices of the various sections of this chapter.

Supported Types:

1. Here are some types and structures defined by the kernel for general use.

```
/*
 *   type of local name of objects:
 */

typedef lname char[LNSIZE];   /* includes a null-terminated character */

/*
 *   structure of capability specifications
 */

struct cpspecs {
 lname name;                  /* local name of the protected object   */
 lname coop;                  /* local name of the cooperation class  */
 char  kind;                  /* type of the object: DIR, MGR, OP, .. */
};
```

2. Here is a list of structures as defined in *"prmstr.h"*.

```
/*
 *   parameter structure used in the Modify primitive
 */

struct prmmod {
 boolean chgname;          /* change capability's name if TRUE */
 boolean chgcoop;          /* change cooperation class if TRUE */
 boolean chgcpcps;         /* change capcaps if TRUE           */
 boolean chgrights;        /* change right if TRUE             */
 lname name;               /* new capability's name            */
 lname coop;               /* new name of cooperation class    */
```

```
 unsigned short capcaps;   /* new capcaps                          */
 unsigned short rights;    /* new rights                           */
};


/*
 *  parameter structure used in the Merge primitive
 */

struct prmmrg {
 CPSPECS mrgcaps[2];       /* first and second capabilities' specs */
 char ctype[2];            /* first and second capabilities' kind  */
};


/*
 *  parameter structure used in View Node primitive
 */

struct prmvwnd {
 char query;          /* view option code: all or partial */
 int  bufsize;        /* size of the provided view buffer */
 lname opname;        /* name of operation; applicable for manager view */
};


/*
 *  parameter structure used in the Compare primitive
 */

struct prmcmp {
   CPSPECS cmpcaps[2];    /* first and second capabilities' specs   */
   char ctype[2];         /* their types: stable or transient       */
   char compare;          /* kind of comparison                     */
};


/*
 *  result structures used in View primitive; and
```

```
 *  parameter structures used in Create primitive
 */

struct dirbuf {              /* directory buffer header           */
 char kind;                  /* view option code                  */
 lname name;                 /* name of the directory             */
 int capcnt[NUM_TRNSLSTS];   /* length of capability lists        */
};

struct mgrbuf {              /* manager buffer header                        */
  lname name;                /* name of the manager                          */
  lname file;                /* name of the object file (CC capability)      */
  boolean depend;            /* interactive/dependency flag                  */
  char protocol;             /* manager instatiation protocol                */
  lname initdir;             /* manager initial active directory             */
  lname dircc;               /* initial active directory coop. class         */
  int numofop;               /* number of exported operations                */
};

struct opnbuf {              /* operation specs buffer header     */
  lname genname;             /* operation's generic name          */
  char lnktype;              /* supporting port type              */
  int prminobj;              /* number of parameters in object    */
  int capinobj;              /* number of capabilities in object  */
  int objsize;               /* object size in bytes              */
  int prmindtl;              /* number of capabilities in details */
  int capindtl;              /* number of parameters in details   */
  int dtlsize;               /* details size in bytes             */
};

struct dt_node {             /* data descriptor node              */
int size;                    /* size of the data in bytes         */
char kind;                   /* type of the data: DIR, MGR, etc.  */
};

struct capbuf {              /* capability buffer                 */
 lname name;                 /* name of the capability            */
 lname coop;                 /* cooperation class                 */
 char kind;                  /* type of the capability            */
```

```
    unsigned short capcaps;   /* capcaps                          */
    unsigned short rights;    /* rights                           */
   };
```

Constants:

Following is a list of constants and their values as given in the *"knldef.h"*.

```
/*
 *  struct names abbreviations
 */

#define DT_NODE   struct dt_node
#define CPSPECS   struct cpspecs

/*
 *  struct size constants
 */

#define    INTSIZE     sizeof (int)
#define    LNSIZE      sizeof (lname)
#define    CPSPECSIZE  sizeof (struct cpspecs)
#define    DTNDSIZE    sizeof (struct dt_node)

#define    PRMMODSIZE  sizeof (struct prmmod)
#define    PRMMRGSIZE  sizeof (struct prmmrg)
#define    PRMVWNSIZE  sizeof (struct prmvwnd)
#define    DIRBUFSIZE  sizeof (struct dirbuf)
#define    CAPBUFSIZE  sizeof (struct capbuf)
#define    MGRBUFSIZE  sizeof (struct mgrbuf)
#define    OPNBUFSIZE  sizeof (struct opnbuf)

/*
 *   numerical constants
 */

#define  FALSE  0
#define  TRUE   1
#define  OFF    0
#define  ON     1
```

```
/*
 *  character/string constants
 */

#define WILDCARD    ""          /* empty string              */
#define CR          '\n'        /* CR: carriage return       */
#define EOS         '\0'        /* end of string character   */
#define EQUAL       0           /* string comparison result  */


/*
 *   manager initiation protocols
 */

#define CREATE      'R'         /* creative                  */
#define CONSRV      'O'         /* conservative              */
#define CCONSRV     'C'         /* class conservative        */

/*
 *   definitions of right bits (position and value)
 */

#define NUM_RIGHTS    16        /* number of rights */

#define BP_COPY_RGT    0        /* copy right                */
#define BP_RGSTR_RGT   1        /* register right            */
#define BP_HOLD_RGT    2        /* hold right                */
#define BP_RM_RGT      3        /* remove right              */
#define BP_VWCP_RGT    4        /* view capability right     */
#define BP_VWND_RGT    5        /* view node right           */
#define BP_MOD_RGT     6        /* modify capability right   */
#define BP_MODND_RGT   7        /* modify node right         */
#define BP_DMGR_RGT    8        /* destroy manager right     */
#define BP_DDIR_RGT    9        /* destroy directory right   */
#define BP_MRG_RGT    10        /* merge right               */
#define BP_TRANS_RGT  11        /* transfer right            */
#define BP_CRTPT_RGT  12        /* create port right         */
#define BP_CDIR_RGT   13        /* change-directory right    */
```

```
#define BP_CRTOP_RGT  14      /* create-operation right  */
#define BP_ASSOC_RGT  15      /* associate-right         */

#define COPY_RGT      0x0001  /* copy right              */
#define RGSTR_RGT     0x0002  /* register right          */
#define HOLD_RGT      0x0004  /* hold right              */
#define RM_RGT        0x0008  /* remove right            */
#define VWCP_RGT      0x0010  /* view capability right   */
#define VWND_RGT      0x0020  /* view node right         */
#define MOD_RGT       0x0040  /* modify capability right */
#define MODND_RGT     0x0080  /* modify node right       */
#define DMGR_RGT      0x0100  /* destroy manager right   */
#define DDIR_RGT      0x0200  /* destroy directory right */
#define MRG_RGT       0x0400  /* merge right             */
#define TRANS_RGT     0x0800  /* transfer right          */
#define CRTPT_RGT     0x1000  /* create port right       */
#define CDIR_RGT      0x2000  /* change-directory right  */
#define CRTOP_RGT     0x4000  /* create-operation right  */
#define ASSOC_RGT     0x8000  /* associate right         */

/*
 *   definitions of capcap bits (position and value)
 */

#define NUM_CAPCAPS  13       /* number of capcaps */

#define BP_COPY       0       /* copy capcap               */
#define BP_RGSTR      1       /* register capcap           */
#define BP_HOLD       2       /* hold capcap               */
#define BP_RM         3       /* remove capcap             */
#define BP_VWCP       4       /* view capability capcap    */
#define BP_VWND       5       /* view node capcap          */
#define BP_MODCP      6       /* modify capability capcap  */
#define BP_MODCPCP    7       /* modify capcap capcap      */
#define BP_MODND      8       /* modify node capcap        */
#define BP_DND        9       /* destroy cd-node capcap    */
#define BP_MRG        10      /* merge capcap              */
#define BP_TRANS      11      /* transfer capcap           */
#define BP_ASSOC      12      /* associate capcap          */
```

```
#define COPY        0x0001      /* copy capcap                */
#define RGSTR       0x0002      /* register capcap            */
#define HOLD        0x0004      /* hold capcap                */
#define RM          0x0008      /* remove capcap              */
#define VWCP        0x0010      /* view capability capcap     */
#define VWND        0x0020      /* view node capcap           */
#define MODCP       0x0040      /* modify capability capcap   */
#define MODCPCP     0x0080      /* modify capcap capcap       */
#define MODND       0x0100      /* modify node cap            */
#define DND         0x0200      /* destroy cd-node capcap     */
#define MRG         0x0400      /* merge capcap               */
#define TRANS       0x0800      /* transfer capcap            */
#define ASSOC       0x1000      /* associate capcap           */

/*
 *   capability type codes
 */

#define ALL         'A'   /* used as default value         */
#define UNKNOWN     'U'   /* not yet specified             */
#define NOCAP       'N'   /* no privilege data             */
#define OPN         'O'   /* operation capability          */
#define DIR         'D'   /* subdirectory capability       */
#define MGR         'M'   /* manager defn capability       */
#define CC          'C'   /* coop-class capability         */
#define SND         'S'   /* Send port capability          */
#define RCV         'R'   /* Receive port capability       */
#define SR          'X'   /* SendReceive port capability   */
#define PTS         'P'   /* unspecified port capability   */

/*
 *  capability kind codes
 */

/*               'U'       is used for UNKNOWN                 */
#define STABLE   'S'   /* stable capability                    */
#define TRNSNT   'T'   /* transient capability                 */
#define TRNSOWN  'O'   /* transient owned capability           */
```

```
#define TRNSBRW   'B'   /* transient borrowed capability       */
#define UOWN      'P'   /* client-end port owned capability    */
#define SOWN      'V'   /* server-end port owned capability    */
#define UBRW      'R'   /* client-end port borrowed capability */
#define SBRW      'X'   /* server-end port borrowed capability */
#define INTENT    'I'   /* is in intention list                */
#define CLIST     'L'   /* c-list                              */
#define ACTDIR    'A'   /* actidir capability                  */

/*
 *   Option codes for primitive ViewNode Manager
 */

#define SPEC     'S'    /* view a specific operation         */
#define BRIEF    'B'    /* view a list of exported operations */

/*
 *   option codes for Compare primitive
 */

#define CMPALL    'A'    /* compatibility test           */
#define CMPCAP    'C'    /* cd-node comparison           */
#define CMPCOOP   'O'    /* cooperation class comparison */

/*
 *   names for different capability lists in a process protection domain.
 *   this constants are useful while manipulating the buffer returned by
 *   ViewNode Directory primitive.
 */

#define NUM_STBLSTS   4    /* number of capability lists in a directory   */
#define NUM_TRNSLSTS  20   /* number of capability lists in a c-list      */
#define NUM_PTLSTS    12   /* number of port-capability lists in a c-list */
#define PTLST_BEGIN   8    /* starting position of port-capability lists  */
#define PTLST_END     20   /* ending position of port-capability lists    */
#define NUM_OWNLSTS   4    /* number of own capability lists in a c-list  */

                         /* position of the stable capability lists     */
#define ACT_SD        0  /* position of the list of directory caps      */
```

```
#define ACT_MG      1   /* position of the list of manager caps        */
#define ACT_OP      2   /* position of the list of operation caps      */
#define ACT_CC      3   /* position of the list of coopclass caps      */

                        /* position of the transient capability lists  */
#define OWN_SD      0   /* position of the list of own directory caps  */
#define OWN_MG      1   /* position of the list of own manager caps    */
#define OWN_OP      2   /* position of the list of own operation caps  */
#define OWN_CC      3   /* position of the list of own coop-class caps */

#define BRW_SD      4   /* position of the list of borrow directory caps  */
#define BRW_MG      5   /* position of the list of borrow manager caps    */
#define BRW_OP      6   /* position of the list of borrow operation caps  */
#define BRW_CC      7   /* position of the list of borrow coop-class caps */

        /* position of the port capability list in c-list */
#define UOWN_SND    8   /* position of the list of client-own send caps   */
#define UOWN_RCV    9   /* position of the list of client-own recv. caps  */
#define UOWN_SDRV   10  /* position of the list of client-own sndrcv caps */

#define SOWN_SND    11  /* position of the list of server-own send caps   */
#define SOWN_RCV    12  /* position of the list of server-own recv. caps  */
#define SOWN_SDRV   13  /* position of the list of server-own sndrcv caps */

#define UBRW_SND    14  /* position of the list of client-brw send caps   */
#define UBRW_RCV    15  /* position of the list of client-brw recv. caps  */
#define UBRW_SDRV   16  /* position of the list of client-brw sndrcv caps */

#define SBRW_SND    17  /* position of the list of server-brw send caps   */
#define SBRW_RCV    18  /* position of the list of server-brw recv. caps  */
#define SBRW_SDRV   19  /* position of the list of server-brw sndrcv caps */
```

Errors:

Here is the complete list of all the error codes and values returned by the primitives as given in *"gtnerr.h"*:

-200	[TCAP EX]	transient capability exist
-201	[SCAP_EX]	stable capability exists
-202	[CAP EX]	capability exists
-220	[NO_MGRCP]	no manager capability
-221	[NO DIRCP]	no directory capability
-222	[NO_OPCP]	no operation capability
-223	[NO CCCP]	no coop-class capability
-224	[NO_PTCP]	no port capability
-225	[NO ASSOCCP]	no associate coop capability
-226	[NO_CAP]	no capability
-227	[NO TRNSCP]	no transient capability
-228	[NO_STBLCP]	no stable capability
-240	[NO COPY RGT]	no copy right
-241	[NO_RGSTR_RGT]	no register right
-242	[NO CDIR RGT]	no change-directory right
-243	[NO_CRTOP_RGT]	no create-operation right
-244	[NO RM RGT]	no remove right
-245	[NO_MOD_RGT]	no modify right
-246	[NO MODND RGT]	no modify node right
-247	[NO_VWCP_RGT]	no view capability right
-248	[NO VWND RGT]	no view node right
-249	[NO_HOLD_RGT]	no hold right
-250	[NO DMGR RGT]	no destroy manager right
-251	[NO_DDIR_RGT]	no destroy subdir right
-252	[NO_CRTPT_RGT]	no create port right
-253	[NO_MRG_RGT]	no merge right
-254	[NO_ASSOC_RGT]	no associate right
-260	[NO_COPY]	no copy capcap
-261	[NO TRANS]	no transfer capcap
-262	[NO RGSTR]	no register capcap
-263	[NO MODCP]	no modify capability capcap
-264	[NO_MODCPCP]	no modify capcap capcap

-265	[NO_MODND]	no modify node capcap
-266	[NO RM]	no remove capcap
-267	[NO_VWCP]	no view capability capcap
-268	[NO VWND]	no view node capcap
-269	[NO_HOLD]	no hold capcap
-270	[NO DND]	no destroy is-node capcap
-271	[NO_MRG]	no merge capcap
-272	[NO ASSOC]	no associate capcap
-290	[INCOMP_CAPS]	incompatible capabilities
-291	[DSTRD DIR]	destroyed directory
-292	[DSTRD_ADIR]	destroyed active directory
-293	[DSTRD MGR]	destroyed manager is-node
-294	[BRWCAP]	borrowed capability
-300	[INVPRM]	invalid input parameters
-301	[BUF_OFLOW]	buffer overflow
-302	[NOSPACE]	kernel out of space
-303	[LOC_NOSPACE]	caller out of space
-310	[NO PNDRQST]	no pending request
-311	[NO_NEWPT]	no port on unaccept list
-312	[INV SNDCAPS]	invalid send capabilities
-313	[INV_SRVCAPS]	invalid send recv caps
-314	[INV OSIZE]	object size is different from the defined size
-315	[USRNOTACS]	user not accessible
-316	[SVRNOTACS]	server not accessible
-317	[DSTR_PND]	destroy-pending
-318	[RECV PND]	receive-pending
-319	[DSTRD_PT]	destroyed port
-320	[UNACK CAPS]	unacknowledge send caps
-321	[OUTSTD_CAPS]	outstanding capabilities
-322	[INVALID CALL]	invalid call for port type
-323	[PT_RFUSD]	request has been refused
-324	[NOT PTORGSVR]	not port server owner
-325	[NOT_PTOWNER]	not port client owner

Primitive:

acptrqst – – accept-request

Calling Convention:

acptrqst *(port, operation, group, sync, status)*
CPSPECS **port, *group;*
lname *operation;*
int *sync;*
int **status;*

Description:

This primitive is invoked by the server of a port to obtain access to a newly created port for servicing the ***operation*** associated with the ***group*** cooperation class. Newly created ports are maintained in a queue. If ***operation*** and ***group*** are set to WILDCARD, the next port from the queue is returned; otherwise the first port that matches the specified ***operation*** and/or ***group*** is returned. The server-end port capability of the accepted port and the associated cooperation class capability are stored in the c-list of the caller under the same name which is specified in ***port***. The port name is also returned in the parameter ***group***.

The returned cooperation class capability is a copy of the cooperation class capability used in the creation of the port (see **CREATE(rts)**), and it exists as long as the port is active. The cooperation class capability associated with a port has the semantics of a capability transferred over a SendReceive port (see **SENDRECEIVE(rts)**): it cannot be destroyed, transferred over a Send port, or registered without copy. When the server-end of a port capability is transferred over another port, its associated cooperation class capability is also transferred.

If ***sync*** is TRUE, the caller blocks until a port supporting the specified ***operation*** and ***group*** is created. ***Status*** contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[LOC_NOSPACE] caller out of space.

[NO_CCCP] invalid cooperation class.

[NO NEWPT] no new ports attached.

[NOSPACE] kernel out of space.

[TCAP_EX] transient capability exists; name duplication.

Command:

assoc   associates a capability with a cooperation class.

Calling Convention:

```
assoc (name, stable, coop, status)
CPSPECS *name;
boolean stable;
lname coop;
int *status;
```

Description:

This primitive associates the *name* directory, manager or operation capability with the *coop* cooperation class capability. If *stable* is TRUE (FALSE), then a stable (transient) capability is associated with the cooperation class.

The new association overwrites the existing association. If no cooperation class capability is specified, the specified capability is disassociated and its cooperation class attribute is unspecified.

The Associate capcap must be active in the participating capabilities. In the case that a stable capability is involved in the association, the caller must have the Associate right for its active directory.

The parameter *status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[CAP_EX] capability exists.

[DSTRD_ADIR] destroyed active directory.

[INVALID_CALL] invalid capability type.

[LOC_NOSPACE] caller out of space.

[NO_ASSOC] no associate capcap.

[NO_ASSOC_RGT] no associate right.

[NO_CAP] no capability.

[NO_CCCP] no cooperation class capability.

[NOSPACE] kernel out of space.

Primitive:

chgdir - - change to a new active directory

Calling Convention:

```
chgdir (dir, status)
CPSPECS *dir;
int *status;
```

Description:

This primitive changes the caller's active directory to the directory specified by *dir*. The directory rights stored in the *dir* capability become the caller's rights for its new active directory. Acquired rights override the capcaps of individual capabilities registered in a directory.

The caller must possess a cooperation class capability which corresponds to the cooperation class associated with the used directory capability. If the used capability is stable, the caller must also have the Change-Directory right for its active directory.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[NO_DIRCP] no directory capability.

[NO_CCCP] no cooperation class capability.

[NO_CDIR-RGT] no change-directory right.

[DSTRD_DIR] directory has been destroyed.

[LOC NOSPACE] caller out of space.

Primitive:

cmpcp -- compare two capabilities

Calling Convention:

```
cmpcp (args, result, status)
prmcmp *args;
int *result;
int *status;
```

Description:

This primitive compares two capabilities, specified in *args*, of same type (directory, manager, operation, cooperation class or port) to check if they are compatible, or point to the same is-node or belong to the same cooperation class.

Args holds the specs of the capabilities to be compared as well as their kind, stable or transient. The structure of *args* is the following:

```
struct prmcmp {
   CPSPECS cmpcaps[2];      /* first and second capabilities' specs    */
   char ctype[2];           /* their types: stable or transient        */
   char compare;            /* kind of comparison                      */
};
```

The *compare* field in *args* designates the attributes of the capabilities to be compared. If *compare* is CMPALL, the primitive makes a compatibility check by comparing if the capabilities point to the same is-node and belong to the same cooperation class. If *compare* is CMPCAP, it checks only if the capabilities point to the same is-node. If *compare* is CMPCOOP, it checks only if the capabilities belong to the same cooperation class.

For cooperation class and port capabilities only CMPCAP option is applicable. Further, it is not is-node, but Channel Control Block, that is compared for ports.

For operation capabilities, the generic names are also checked for comparisons COMP_ALL and COMP CAP.

If the comparison is successful, then *result* returns TRUE else it returns FALSE. In the case of port capabilities, this primitive returns TRUE, if the two capabilities are the client- and the server-end port capabilities of the same port.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[INVALID CALL] capabilities are not of the same type.

[LOC NOSPACE] caller out of space.

[NO_CAP] one or both of the specified capabilities do not exist.

[NOSPACE] kernel out of space.

Command:

copy – – copy a capability

Calling Convention:

```
copy (name, newname, ckind, status)
CPSPECS *name, *newname;
char ckind;
int *status;
```

Description:

This primitive copies a stable (transient) capability and stores it in the caller's active directory (c-list).

This primitive expects the type of the capability to be directory, manager, operation or cooperation class. The Copy capcap must be active in the capability to be copied. If a stable capability is copied, the caller must have, in addition, the Copy right for its active directory,.

Name and *newname* are the specifications (name, cooperation class and type) of the old and new capability. The parameter *ckind* can take the values TRNSNT or STABLE. *Status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD ADIR] destroyed active directory.

[INCOMP CAPS] incompatible capabilities.

[INVPRM] invalid input parameters.

[LOC_NOSPACE] caller out of space.

[NO_COPY] no copy capcap.

[NO_COPY_RGT] no copy right.

[NOSPACE] kernel out of space.

Primitive:

crtdir – – create directory

crtmgr – – create manager

crtop – – create operation

crtcc – – create cooperation-class

crtpt – – create port

Calling Convention:

```
crtdir (name, status)
CPSPECS *name;
int *status;

crtmgr (name, params, prmsize, status)
CPSPECS *name;
char *params;
int prmsize;
int *status;

crtop (name, manager, generic, status)
CPSPECS *name, *manager;
lname generic;
int *status;

crtcc (name, status)
CPSPECS *name;
int *status;

crtpt (name, operation, group, status)
CPSPECS *name, *operation, *group;
int *status;
```

Description:

These primitives create an object (directory, manager, operation, cooperation-class or port) and a corresponding capability. The created capability is stored in the caller's c-list. All the capcaps/rights of all capabilities except that of the (client-end) port capability are active. The capcaps of the (client-end) port capability are set according to the capcaps of the operation capability used in the creation of the port. *Status* contains the outcome of the invocation.

The `crtdir` directive creates a new directory called *name*.

The **crtmgr** directive creates a new manager called *name*. The *params* argument points to the parameter buffer of the call which contains the attributes of the manager. The *prmsize* designates the size in bytes of the parameters (*params* buffer). The parameter buffer holds the general attributes of the manager followed by list of the specifications of the exported operations.

The general specifications of a manager are:

the manager's local name (**lname**);
the name of the coop-class capability identifying the image file (**lname**);
the flag indicating that the manager is interactive (1) or not (0) (**short**);
the manager's lifetype or instantiation protocol (**char**);
the local name of initial directory (**lname**);
the cooperation class to which the initial directory belongs (**lname**);
the number of exported operations (**int**);

The caller must possess the cooperation class and directory capabilities to be used in the creation of the manager. The lifetype can take the values of the constants **CREATE**, **CONSRVT** (conservative), or **CCONSRV** (class conservative).

The specifications of each operation are:

the operation's (generic) name known to the manager (**lname**);
the type of the port to be used for requesting the operation (**char**);
the number of parameters in the sent/received message (**int**);
the number of capabilities in the sent/received message (**int**);
the message size in bytes (**int**);
the number of parameters in the details (**int**);
the number of capabilities in the details (**int**);
the details size in bytes (**int**);
/* message's first parameter description */
the size of the parameter in bytes (**int**);
the type of the parameter (**char**);
/* message's second parameter description */
. . .
/* details' first parameter description */
the size of the parameter in bytes (**int**);
the type of the parameter (**char**);
/* details' second parameter description */
. . .

The port type can take the values: **SND** (Send), **RCV** (Receive) or **SR** (SendReceive). The values

for the type of the parameters in message/details descriptors are: DIR (directory), MGR (manager), OPN (operation), CC (cooperation class), SND (send), RCV (receive), SR (sendreceive), or NOCAP (no capability data). All the capability data descriptors must preceed the non-capability ones. Each capability data descriptor has size equal to that of **struct cpspecs** (CPSPECS) - capability specification.

The following structures have been declared for type casting the different segments of the *params* buffer, making its manipulation easy (see **INTRO(rts)**):

> struct mgrbuf for the general specifications of a manager.
> struct opnbuf for the general specifications of an operation.
> struct dt_node for data descriptors.

The crtop directive creates a new operation called ***name*** which corresponds to the ***generic*** operation exported by the ***manager***. This directive requires the possession of a manager capability to the specified ***manager*** and a cooperation class capability that corresponds to the cooperation class associated with the used manager capability. The caller is also required to have the CREATE-OPERATION right for its active directory in case a stable manager capability is used.

The crtcc directive creates a new cooperation class called ***name***.

The crtpt directive creates a port called ***name*** of the appropriate message and port type to support the request of the specified ***operation***. *Group* designates the cooperation class with which the port is to be associated. *Group* should be either the cooperation class associated with the used operation capability, if such association has been specified; or ***group*** should specify a cooperation class capability.

The port creation generates two port capabilities, one for accessing the client-end of the port and the other the server-end of the port. A copy of *group* cooperation class capability is also created. The client-end port capability is stored in the caller's c-list. The server-end port capability and the cooperation class capability are delivered to the server of the port when the server invokes the acptrqst primitive (ACCEPT-REQUEST(rts)).

If a stable operation capability is used in the creation of the port, the caller must have the CREATE-PORT right for its active directory.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD MGR] destroyed manager.

[INVPRM] invalid parameter.

[LOC_NOSPACE] caller out of space.

[NO_CCCP] cooperation class capability is not found.

[NO_DIRCP] directory capability is not found.

[NO_MGRCP] manager capability is not found.

[NO_OPCP] operation capability is not found.

[NO_CRTOP_RGT] create operation right is not active.

[NO_CRTPT_RGT] create port right is not active.

[NOSPACE] kernel out of space.

[TCAP EX] a transient capability already exists.

Remarks:

The flag indicating whether a manager is interactive or not is required by the current implementation of the Gutenberg system and it is not part of the design specifications. When a tty-manager is written, this parameter shall be dropped from the general attributes of a manager.

Primitive:

dstnd - - destroy a node in the Interconnection Schema

dstpt - - destroy a port

Calling Convention:

```
dstnd (name, status)
CPSPECS *name;
int *status;

dstpt (name, status)
CPSPECS *name;
int *status;
```

Description:

This primitive destroys the *name* object (directory, manager, or port). *Status* contains the outcome of the invocation.

Dstnd directive destroys the *name* node of the Interconnection Schema. The used capability is also destroyed with the destruction of the node, unless it is an outstanding capability over a SendReceive port (see **SENDRECEIVE**(rts)). This directive requires that the caller has the Destroy-Directory-Node or Destroy-Manager-Node right for its active directory, if a stable capability is used in the destruction of a directory or manager node respectively. In any case, the Destroy-Node capcap must be active in the capability.

Dstpt directive destroys the *name* port, the two corresponding port-end capabilities and the associated cooperation class capability. It can only be invoked by the owner of the port who should possess the port capability at the time of the invocation. All pending requests are discarded and any dangling capabilities that are part of the requests are returned to the sender. In the case of SendReceive port in which there is a possibility of outstanding capabilities, the system waits for the request in progress to complete before destroying the port. The port to be destroyed is marked *destroy-on-clear* and when the server of the port performs the SEND, the kernel revokes the outstanding capabilities, ignores the sent message and finalizes the destruction of the port. The case of the server-end capability being outstanding over a SendReceive port is handled in the same way.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD PND] destroyed pending.

[INVPRM] invalid capability type is specified.

[LOC_NOSPACE] caller out of space.

[NO_CAP] no capability.

[NO_DND] no destroy is_node capcap.

[NO_DDIR_RGT] no destroy subdir is_node right.

[NO_DMGR_RGT] no destroy manager is_node right.

[NOT_PTONWER] caller not the port owner; caller cannot destroy the port.

Primitive:

drpcp – – drop a transient capability from c-list

Calling Convention:

```
drpcp (name, status)
CPSPECS *name;
int *status;
```

Description:

This primitive removes the *name* capability (directory, manager, operation or cooperation class) from the caller's c-list, unless the capability was previously received over a SendReceive port (see **SENDRECEIVE(rts)**). In the case of directory, manager or operation capability, the corresponding Interconnection Schema node is destroyed, if the dropped capability is the last one linked to that node.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[BRWCAP] capability received over a SendReceive port.

[LOC NOSPACE] caller out of space.

[NO TRNSCP] no transient capability.

Primitive:

exam -- examine the messages/details of a port

Calling Convention:

```
exam (name, nofmsg, exambuf, exbsize, status)
CPSPECS *name;
int *nofmsg;
char *exambuf;
int *exbsize;
int *status;
```

Description:

This primitive allows the server of the *name* port to examine the messages on the port without removing them; or to find out the number of messages on the port.

Nofmsg contains the number of messages to be examined. If *nofmsg* is zero, then the number of messages on the port is returned in the *nofmsg*; otherwise *nofmsg* contains the actual number of the returned messages which is less or equal to the requested number of messages.

Exambuf contains the returned messages. In case of buffer overflow, *exambuf* contains an integer which is the required buffer size in bytes.

Exbsize contains always the size of *exambuf* in bytes.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[BUF OFLOW] buffer overflow.

[DSTRD PT] port has been destroyed.

[LOC NOSPACE] caller out of space.

[NO PTCP] no port capability.

[NOSPACE] kernel out of space.

[USRNOTACS] user of the port not accessible.

Remarks:

The current implementation of exam returns the names and not the content of the capabilities which are part of the messages, as the vwcp (**VIEW(rts)**).

Primitive:

getdtls – – get details

Calling Convention:

getdtls (name, dtlsbuf, dtlssize, sync, status)

`CPSPECS` **name*;

`char` **dtlsbuf*;

`int` **dtlssize*;

`int` *sync*;

`int` **status*;

Description:

This primitive removes the request details from the SendReceive port ***name*** and stores them in the provided *dtlsbuf* buffer. This primitive can only be invoked by the server of the SendReceive port.

If capabilities are expected as part of the request details, their specifications (name, cooperation class and type) must appear in the *dtlsbuf*. The order of the parameters in *dtlsbuf* should be the same as in the specifications of the corresponding operation (see `crtmgr` directive in **CREATE**(rts)). The specifications of the expected capabilities must be unique in the caller's clist.

The transferring of capability as part of the *details* is temporary and the transferred capability is held only while the server is processing the request which pertains to. When the server executes the send that satisfies the request, the kernel automatically returns the outstanding capabilities to the client. Outstanding capabilities cannot be destroyed, further transferred over a Send port (using send) or registered without copy. Before returning to the sender, modified outstanding capabilities are restored to the state they were in at the time of receive.

Dtlssize contains the size of the *dtlsbuf* in bytes, and must be in agreement with the message size defined by the specifications of the corresponding operation.

If *sync* is `TRUE`, the caller blocks until a sndrcv (**SENDRECEIVE(rts)**) primitive is invoked on the port; otherwise the caller is not blocked while the `getdtls` is pending (asynchronous receive). *Status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD PT] port has been destroyed.

[INVALID CALL] invalid call.

[INV_OSIZE] invalid size of message/details.

[LOC_NOSPACE] caller out of space.

[NOSPACE] kernel out of space.

[NO_PTCP] no port capability.

[RECV_PND] receive pending.

[TCAP_EX] transient capability exists; name duplication.

[USRNOTACS] user of the port not accessible.

Primitive:

hldcp - - make a stable capability, transient

Calling Convention:

```
hldcp (name, copy, status)
CPSPECS *name;
boolean copy;
int *status;
```

Description:

This primitive adds the *name* stable capability to the caller's c-list. If *copy* is TRUE, then a copy of the stable capability is added to the c-list; otherwise, the stable capability is removed from the caller's active directory and added on the c-list.

The caller must have the Hold right for its active directory. The Hold capcap must be active in the capability. If *copy* is TRUE, the caller must also have the Copy right for its active directory, and the Copy capcap must be active in the capability.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_ADIR] destroyed active directory.

[LOC_NOSPACE] caller out of space.

[NO_COPY] no copy capcap.

[NO_COPY_RGT] no copy right.

[NO STBLCP] no stable capability.

[NO HOLD] no hold capcap.

[NO HOLD RGT] no hold right.

[NOSPACE] kernel out of space.

[TCAP_EX] transient capability exists.

Primitive:

homedir -- - change to the login or primary directory

Calling Convention:

homedir (status)
int **status*;

Description:

This primitive changes the caller's active directory to its login or primary directory. *Status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[LOC_NOSPACE] caller out of space.

Primitive:

mrgcp – – merge a capability with another compatible capability

Calling Convention:

```
mrgcp (args, status)
struct prmmrg *args;
int *status;
```

Description:

This primitive finds the union of the capcaps and the union of rights of two compatible capabilities and assigns them to the corresponding fields of the first capability. Two capabilities are compatible if they are of the same type, point to the same node of the Interconnection Schema and belong to the same cooperation class.

This primitive expects the type of the capabilities to be directory, manager, operation or cooperation class. The Merge capcap must be active in the participating capabilities. If a stable capability is to be merged, the caller must have, in addition, the Merge right for its active directory.

Args holds the specs of the participating capabilities as well as their kind, stable or transient. The structure of *args* is the following:

```
struct prmmrg {
CPSPECS mrgcaps[2];
char ckind[2];
};
```

Ckind can take the values `STABLE`, for stable or `TRNSNT`, for transient.

The parameter *status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[CAP_EX] capability exists; duplication of name/cooperation.

[DSTRD ADIR] destroyed active directory.

[INCOMP CAPS] incompatible capabilities.

[INVPRM] invalid input parameters.

[LOC NOSPACE] caller out of space.

[NO_CAP] no capability.

[NO_MERGE] no merge capcap.

[NO_MERGE_RGT] no merge right.

[NOSPACE] kernel out of space.

Primitive:

modcp – – modify the fields of a transient capability

Calling Convention:

```
modcp (name, ckind, args, status)
CPSPECS *name;
char ckind;
struct prmmod *args;
int *status;
```

Description:

This primitive modifies the attributes of the capability *name* of the specified kind *ckind* (ACTDIR: active directory, STABLE: stable or TRNSNT: transient). The attributes which can be modified are the local names of the capability and its associate cooperation class, the capcaps and the rights. These are specified in the *args*. The type of *args*, *struct prmod*, is defined as follows:

```
struct prmmod {
boolean chgname; /* new capability name flag */
boolean chgcoop; /* new cooperation class name flag */
boolean chgcpcps; /* new capcaps flag */
boolean chgrights; /* new rights flag */
lname name; /* new local name of the capability */
lname coop; /* new local name of associate cooperation class */
unsigned short capcaps; /* new capcaps */
unsigned short rights; /* new rights */
};
```

If *ckind* is STABLE, then the caller must have the Modify right for its active directory to modify any attribute of the capability. In all other cases, if the capcaps of the capability are to be modified, the Modify-Capcap capcap must be active in the capability, and if the other attributes of the capability are to be modified, the Modify-Capability capcap must be active.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[CAP EX] capability exists; duplication of name/cooperation.

[DSTRD ADIR] destroyed active directory.

[LOC_NOSPACE] caller out of space.

[NO_CAP] no capability.

[NO_MODCP] no modify capability capcap.

[NO_MODCPCP] no modify capcap capcap.

[NO_MOD_RGT] no modify right.

Primitive:

qrypts - - query ports

Calling Convention:

```
qrypts (portlist, listlen, sync, status)
char *portlist;
int *listlen;
int sync;
int *status;
```

Description:

This primitive queries the set of existing ports and returns those that have any pending requests or destroy-on-clear status (see **DESTROY(rts)**).

Portlist points to a buffer which will hold the returned list of ports. *Listlen* initially holds the size of the provided buffer, *portlist,* in bytes. In case of successful completion, *portlen* contains the actual size of the returned *portlist.*

If *sync* is TRUE, the caller blocks waiting for a port event. *Status* contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[BUF OFLOW] buffer overflow.

[LOC NOSPACE] caller out of space.

[NO_PNDRQST] no pending request.

[NOSPACE] kernel out of space.

Primitive:

recv -- receive a message over a port

Calling Convention:

```
recv (name, rcvbuf, rcvsize, sync, status)
CPSPECS *name;
char *rcvbuf;
int *rcvsize;
int sync;
int *status;
```

Description:

This primitive removes the next message from the *name* port and stores it in the specified *rcvbuf* buffer of *rcvsize* size (in bytes). If capabilities are expected as part of the message, their specifications (name, cooperation class and type) must be given in the *rcvbuf*. The order of the parameters in *rcvbuf* should be the same as defined in the specifications of the corresponding operation (see `crtmgr` directive in **CREATE**(rts)). The capability specifications must be unique in the caller's clist. *Rcvsize* contains the size of the *rcvbuf* in bytes, and must be in agreement with the message size defined by the specifications of the corresponding operation.

If *sync* is `TRUE`, the caller is blocked until a send (**SEND**(rts)) primitive is invoked on the port (synchronous `receive`); otherwise the caller is not blocked while the receive is pending (asynchronous `receive`).

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[BUF_OFLOW] buffer overflow.

[DSTRD_PT] port has been destroyed.

[INV_OSIZE] invalid size of message/details.

[INVPRM] invalid parameter.

[LOC_NOSPACE] caller out of space.

[NOSPACE] kernel out of space.

[PT_RFUSD] request has been refused.

[RECV_PND] receive pending.

[SVRNOTACS] server of the port not accessible.

[TCAP_EX] transient capability exists; name duplication.

[USRNOTACS] user of the port not accessible.

Primitive:

refuse -- refuse a request on a port

Calling Convention:

`refuse` (*name, status*)

`CPSPECS` **name*;

`int` **status*;

Description:

This primitive rejects the next request for service on the *name* port as unsatisfiable. This primitive can only be invoked by the server of the port. Any dangling capabilities resulting from the refutation of the request are returned to the sender.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[INVALID CALL] invalid call; nothing to refuse.

[LOC NOSPACE] caller out of space.

[NO PTCP] no port capability.

[USRNOTACS] user of the port not accessible.

Primitive:

regcp – – make a transient capability stable

Calling Convention:

```
regcp (name, copy, status)
CPSPECS *name;
boolean copy;
int *status;
```

Description:

This primitive adds the *name* transient capability in the caller's active directory. If *copy* is TRUE, then a copy of the transient capability is added in the active directory; otherwise, the transient capability is removed from the caller's c-list and added in the active directory. Port capabilities *cannot* be registered.

The caller must have the Register right for its active directory, and the capability to be registered must have its Register capcap active. If *copy* is TRUE, the caller, in addition, must have the Copy right for its active directory, and the Copy capcap must be active in the capability.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_ADIR] destroyed active directory.

[LOC_NOSPACE] caller out of space.

[NO_COPY] no copy capcap.

[NO_COPY_RGT] no copy right.

[NO_RGSTR] no register capcap.

[NO_RGSTR_RGT] no register right.

[SCAP_EX] stable capability exists; name duplication.

[NO_TRNSCP] no transient capability.

Primitive:

reject -- reject port

Calling Convention:

```
reject (name, status)
CPSPECS *name;
int *status;
```

Description:

This primitive rejects the *name* port. This primitive can only be invoked by the original server of the port who owns the server-end capability of the port and it is similar to the dstpt (**DESTROY(rts)**) invoked by the user of the port. All pending requests are discarded and any dangling capabilities resulting from the rejection of the requests are returned to the sender. The corresponding port capabilities of both ends are removed from the user's and server's c-list.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_PND] port destroyed is pending.

[INVALID_CALL] invalid call.

[LOC_NOSPACE] caller out of space.

[NOSPACE] kernel out of space.

[NOT_PTORGSVR] caller not port original server; caller cannot reject the port.

Primitive:

rmvcp – – remove a stable capability from the Interconnection Schema

Calling Convention:

`rmvcp` (*name, status*)

`CPSPECS` **name*;

`int` **status*;

Description:

This primitive removes the *name* capability from the caller's active directory. In the case of directory, manager or operation capability, the corresponding Interconnection Schema node is destroyed, if the removed capability is the last one linked to that node.

The caller is required to have the Remove right for its active directory while the Remove capcap must be active in the capability.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_ADIR] destroyed active directory.

[LOC_NOSPACE] caller out of space.

[NO_RM] no remove capcap.

[NO_RM_RGT] no remove right.

[NO_STBLCP] no stable capability.

[NOSPACE] kernel out of space.

Primitive:

revoke – – revoke a request on a port

Calling Convention:

```
revoke (name, status)
CPSPECS *name;
int *status;
```

Description:

This primitive revokes the last request on the *name* port. It can only be invoked by the user of the port. All the dangling capabilities resulting from the revocation of the request are returned to the sender and the message is discarded. A request can be revoked only before it is satisfied by the server.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_PND] port destroyed is pending.

[LOC_NOSPACE] caller out of space.

[NO_PNDRQST] no pending request to be revoked.

[NO_PTCP] no port capability.

[SVRNOTACS] server of the port not accessible.

Primitive:

send - - send a message over a port

Calling Convention:

send (*name, message, msglen, ack, sync, status*)
CPSPECS **name*;
char **message*;
int *msglen*;
int *ack*;
int *sync*;
int **status*;

Description:

This primitive puts the *message* of size *msglen* (in bytes) on the *name* port. The type of the port can be Send in which case the caller must be the user of the port, or Receive or SendReceive in which cases the caller must be the server of the port. In the case of SendReceive port, the kernel automatically transfers the outstanding capabilities in their original form to the client (**SENDRECEIVE**(rts) and **GET-DETAILS**).

If capabilities are transferred as part of the message, they must have their Transfer capcap active and their specifications (name, cooperation class and type) must appear in the *message*. The order of the parameters in the *message* should be the same as in the specifications of the corresponding operation (see crtmgr directive in **CREATE**(rts)). A transferred capability is removed from the possession of the sender if it is an exclusive capability, i.e. a port capability or a transient capability with its Copy capcap inactive.

Msglen contains the size of the *message* in bytes, and must be in agreement with the message size defined by the specifications of the corresponding operation.

If *ack* is TRUE, the sender expects to be acknowledged when its correspondent over the port receives the message. If capabilities are transferred as part of the message, the system requires the *ack* to be set to TRUE. The transfer by **send** is permanent and in the case of transferring of a port, the ownership (privilege to destroy or reject the port) of the port is also passed.

If *sync* is TRUE, the sender blocks waiting for the delivery of the message (synchronous send); otherwise the sender executes concurrently with the delivery of the message (asynchronous send).

At any given time, *status* contains the state of the invocation. The acknowledgement sets the *status* to SUCCESS in the case of acknowledge-send.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD PT] port has been destroyed.

[INVALID CALL] invalid call.

[INV_OSIZE] invalid size of message/details.

[INV_SNDCAPS] a capability to be transferred does not exist or it has its transfer capcap inactive.

[LOC NOSPACE] caller out of space.

[NO PTCP] no port capability.

[NOSPACE] kernel out of space.

[OUTSTD_CAPS] outstanding capabilities are not in a state to be revoked.

[PT_RFUSD] request has been refused.

[RECV_PND] receive pending.

[SVRNOTACS] server of the port not accessible.

[UNACK_CAPS] capabilities sent on a port must be acknowledge.

[USRNOTACS] user of the port not accessible.

Primitive:

sndrcv - - send details over a port and receive a reply message

Calling Convention:

`sndrcv` (***name, details, dtlsize, rcvbuf, rcvsize, sync, status***)

`CPSPECS` **name*;

`char` **details*;

`int` *dtlsize*;

`char` **rcvbuf*;

`int` **rcvsize*;

`int` *sync*;

`int` **status*;

Description:

This primitive puts the *details* of a request on the *name* SendReceive port for the server to use in satisfying the request. When the server responds to the request by executing a **send** **SEND(rts)**), the server's reply is stored in the *rcvbuf* buffer provided by the client. (`sndrcv` has semantics of remote procedure call from caller's point of view). *Dtlsize* and *rcvsize* contain the size (in bytes) of the *details* and *rcvbuf* respectively, and should be in agreement with the defined sizes in the corresponding operation (see `crtmgr` directive in **CREATE(rts)**).

If capabilities are transferred as part of the *details* or expected as part of the reply, their specifications (name, cooperation class and type) must appear in the *details* and/or *rcvbuf*. The order of the parameters in `details` and `rcvbuf` should be the same as in the specifications of the corresponding operation. The capabilities to be passed as part of the details must have their Transfer capcap active. A transferred capability is removed from the possession of the sender if it is an exclusive capability, i.e. a port capability or a transient capability with its Copy capcap inactive.

The transferring of a capability as part of the *details* is temporary and the transferred capability is held only while the server is processing the operation request. When the server executes the **send** that satisfies the request, the kernel automatically returns the outstanding capabilities in their original form to the client. The ownership (destroy or reject privilege) of a port capability can never be passed using the `sndrcv`.

If *sync* is `TRUE`, the caller blocks until the server replies.

Status contains the outcome of the invocation.

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[DSTRD_PT] port has been destroyed.

[INVALID_CALL] invalid call.

[INV_OSIZE] invalid size of message or details.

[INV_SRVCAPS] a capability to be transferred does not exist or it has its transfer capcap inactive.

[LOC_NOSPACE] caller out of space.

[NO_PTCP] no port capability.

[NOSPACE] kernel out of space.

[PT_RFUSD] request has been refused.

[RECV_PND] receive pending.

[SVRNOTACS] server of the port not accessible.

[TCAP_EX] transient capability exists; name duplication.

Primitive:

vwact – – view active directory

vwclst – – view c-list

vwcp – – view a capability

vwnd – – view a node of the Interconnection Schema

Calling Convention:

```
vwact (args, vwbuf, vwsize, status)
struct prmvwnd *args;
char *vwbuf;
int *vwsize;
int *status;

vwclst (args, vwbuf, vwsize, status)
struct prmvwnd *args;
char *vwbuf;
int *vwsize;
int *status;

vwcp (name, vwbuf, vwsize, status)
CPSPECS *name;
char *vwbuf;
int *vwsize;
int *status;

vwnd (name, args, vwbuf, vwsize, status)
CPSPECS *name;
struct prmvwnd *args;
char *vwbuf;
int *vwsize;
int *status;
```

Description:

This primitive brings a copy of the caller's active directory, caller's c-list, the *name* capability or node of the Interconnection Schema (is-node) into the address space of the caller. The copied object is stored in the specified *vwbuf* buffer of size *vwsize* in bytes. In case of overflow, *vwbuf* contains an integer which is the expected buffer size. *Status* contains the outcome of the invocation.

The *vwact* and *vwclst* return a list of the capabilities specifications (name, cooperation class and type) stored in the caller's active directory and c-list. The *vwnd* directive returns a directory or manager is-node. In the case of a directory, a list of all stored capabilities is returned, whereas in the case of manager definition, all the manager's attributes ini-tial active directory, instantiation protocol, associate cooperation class, and a list of all the exported operations, are returned.

All three directives support partial views. For `vwact`, `vwclst`, and `vwnd` directory, only capabilities of a specific type can be returned, such as only directory capabilities. For `vwnd` manager, the attributes of a specific operation can be returned instead of a listing of all the exported operations. The parameters of the view are specified in the *args* whose type is defined as follows:

```
struct prmvwnd {
char query; /* type of view */
int bufsize; /* size of the provided buffer in bytes */
lname opname; /* name of the operation; applicable for manager view */

};
```

When viewing a manager the *query* field can take the values of the constants `SPEC` for returning a specific operation in which case the generic name of the operation should be specified in the *opname* field, or `BRIEF` for brief listing of all operations. The default value is `BRIEF`.

When viewing a directory, including an active one, the *query* field can take any of the four values: `DIR`, `MGR`, `OPN`, or `CC`. In the case of *vwclst*, the following values are also applicable: `SND` for send ports, `RCV` for receive ports and `SR` for sendreceive ports. The default value is `UNKNOWN` which returns a total view of the directory node.

If an active directory, a c-list or a directory node is viewed, the *vwbuf* contains the following information:

the class of returned capabilities; same as the *query* field in *args* (`char`);
the local name of the directory (`lname`);
list of the numbers of the contained capabilities of each type (array of `int`);
list of the specifications of stored capabilities (array of `CPSPECS`);

The order of the capability kinds in the arrays is:

the directory, manager, cooperation class, and operation;
the *borrow* directory, manager, cooperation class, and operation;
the *use-end* send, receive, and sendreceive;
the *borrow use-end* send, receive and sendreceive;
the *serve-end* send, receive and sendreceive; and
the *borrow serve-end* send, receive and sendreceive.

Borrow capabilities are the ones received over a SendReceive port (see **SENDRECEIVE(rts)**).

If a manager node is viewed, the *vwbuf* contains the kind of the request as specified in *query* field of the *args*, followed by the general specifications of the manager, followed by either a list of the exported operation, or the attributes (see, `crtmgr` in **CREATE(rts)**) of a specific operation.

The `vwact`, `vwclst` and `vwnd` directives require that the caller has the View-Node right for its active directory while the View-Node capcap must be active in the capability used.

The *vwcp* directive returns the attributes of a capability. It requires that the caller has the View-Capability right for its active directory, if a stable capability is to be viewed. The View-Capability capcap must be active in the viewed capability.

When a capability is viewed, the *vwbuf* contains the general specifications of a capability which are:

the name of the object that protects (`lname`);
the name of the cooperation class in which the capability belongs (`lname`);
the type of the capability (`char`);
the capcaps word (`unsigned short`);
the rights word (`unsigned short`);

In the case of an operation capability, the general specifications of the capability are followed by the specifications of the operation as in the case of vwnd manager with the SPEC option.

The following structures have been declared to type cast the different segments of the *vwbuf* buffer, making its manipulation easy (see **INTRO(rts)**):

`struct mgrbuf` for the general specifications of a manager.
`struct opnbuf` for the general specifications of an operation.
`struct dt_node` for data descriptors.
`struct capbuf` for the general specifications of a capability.
`struct dirbuf` for the contain of a directory (header of *vwbuf*).

Return Status:

Returns 0 on success, negative number on failure.

Errors:

[BUF_OFLOW] buffer overflow.

[DSTRD_ADIR] destroyed active directory.

[INVPRM] invalid operation name in *args*.

[LOC_NOSPACE] caller out of space.

[NO_CAP] no capability.

[NOSPACE] kernel out of space.

[NO_VWCP] no view capability capcap.

[NO_VWCP_RGT] no view capability right.

[NO_VWND] no view node capcap.

[NO_VWND_RGT] no view node right.

3 Gutenberg Command Processor

3.1 Introduction

This chapter describes the commands of the *gcp*, the *gutenberg command processor*. Gcp is not part of the Gutenberg kernel, but it is a good example of a Gutenberg manager and illustrates how the Gutenberg primitives can be used. Gcp is written to support interactive users. The login process upon successful login starts up a gcp which prompts for a command. Gcp executes commands read from a terminal (xterm in the current implementation) or from a file (script). It recognizes two kinds of commands: *internal* and *external*.

Internal commands allow for direct manipulation of the Interconnection Schema. These are procedures which prompt for and read in the necessary parameters for invoking a kernel primitive.

External commands allow for requesting an operation on an object managed by another process over a SendReceive port. The procedure handling the external commands prompts for the necessary arguments using their description in the corresponding operation capability. It then creates a port for requesting the operation and invokes the **SENDRECEIVE**(rts) primitive to send the details of the request and receive a reply back.

Gcp traps the *control-C* signal (SIGINT) and cancels the execution of the last command. A command can be canceled before the request is related to the kernel.

The calling convention for almost all gcp commands is

```
command obj
```

There are some commands that do not require an *obj*. The possible values for *obj* are `dir` for directory, `mgr` for manager, `op` for operation, `cc` for cooperation class, and `pt` for port. In the case of an external command, the *obj* is interpreted as the name of the cooperation class capability to be used during the creation of the port.

The errors and status which the gcp returns are either generated by the gcp commands or are the ones that the kernel returns as a result of a primitive invocation.

Here is a list of the gcp commands:

ASSOCIATE (`asoct` and `assocs`) associate a capability with a cooperation class.

CHANGE-DIRECTORY (`chgdir` or `cd`) change to a new active directory.

COMPARE (`cmp`) compare two capabilitities.

COPY (`cps` and `cpt`) copy a capability.

CREATE (`crt`) create a component of the Interconnection Schema.

DESTROY (`dst`) destroy a node of the Interconnection Schema.

DROP (`drp`) remove a transient capability from a c-list.

HOLD (`hld` and `hldc`). make stable capability transient.

HOME (`home`) change to the login or primary directory.

LOGOUT (`log`) exit the gcp.

MERGE (`mrg`) merge two compatible capabilities.

MODIFY (`mods`, `modt` and `moda`) modify a stable, a transient or an active directory capability.

REGISTER (`reg` and `regc`) make a transient capability stable.

REMOVE (`rmv`) remove stable capability from an active directory.

SCRIPT (`script` and `quit`) execute a command file.

VIEW (`vwcp`, `vwact`, `vwclst` and `vwnd`) display a capability, active directory, c-list or a node of the Interconnection Schema.

3.2 GCP Commands

ASSOCIATE(gcp)

Command:

asoct – – associates a cooperation class capability with a transient capability

asocs – – associates a cooperation class capability with a stable capability

Calling Convention:

`asoct` *obj*

`asocs` *obj*

Description:

This command associates a directory, manager or operation capability with a cooperation class by calling the **ASSOCIATE** primitive. The new association overwrites the existing association. The command prompts for and reads the name and cooperation class of the capabilities to be associated. If no new cooperation class capability is specified, the specified capability is disassociated and its corresponding cooperation class attribute is unspecified.

`Asocs` (`asoct`) associates a stable (transient) capability. Both directives require that the Associate capcap be active in the participating capabilities. In the case that a stable capability is involved in the association, the caller must have the Associate right for its active directory.

Obj can take the values of `dir`, `mgr` or op. For the return status see **ASSOCIATE(rts)**.

CHANGE-DIRECTORY(gcp)

Command:

chgdir or cd – – change to a new active directory

Calling Convention:

`chgdir`

`cd`

Description:

The command `chgdir` or `cd` prompts for and reads in the name of the directory which should become the new active directory, and the name of the cooperation class associated with the directory capability. If such cooperation class is associated with the directory capability, the caller must possess a corresponding cooperation class capability at the invocation time.

This command calls the **CHANGE-DIRECTORY** primitive. For the return status see **CHANGE-DIRECTORY(rts)**.

COMPARE(gcp)

Command:

cmp - - compare two capabilities

Calling Convention:

cmp *obj*

Description:

This command prompts for and reads in the names and the cooperation classes of two capabilities of type *obj* and the type of comparison and then it calls the COMPARE primitive.

There are three comparison types: ***compatibility check*** by checking if the capabilities point to the same is-node and belong to the same cooperation class; ***cooperation class check*** by checking whether the capabilities belong to the same cooperation class; and ***object check*** by checking if the capabilities point to the same node of the Interconnection Schema.

Obj can take any of the five values for dir, mgr, op, cc, or pt. For the return status see **COMPARE(rts)**.

COPY(gcp)

Command:

cps - - copy a stable capability

cpt - - copy a transient capability

Calling Convention:

cps *obj*

cpt *obj*

Description:

cps (cpt) copies a stable (transient) capability and stores it in the caller's active directory (c-list), unless another capability of the same type, name and cooperation class already exists in the active directory (c-list).

This primitive requires that the Copy capcap be active in the capability to be copied. If a stable capability is copied, the caller must have, in addition, the Copy right for its active directory,.

Obj can take the values of dir, mgr, op, or cc.

This command prompts for and reads in the names and cooperation classes of the old and new capability. Then, it calls the COPY primitive. For the return status see **COPY(rts)**.

CREATE(gcp)

Command:

crt - - create an object

Calling Convention:

`crt` *obj*

Description:

This command creates an object (directory, manager, operation, or cooperation-class) and a corresponding capability unless another capability of the same type, name and unspecified cooperation class already exists in the caller's c-list. In the case of successful creation, the created capability is stored in the caller's c-list with all the capcaps/rights active.

Obj can take the values of `dir`, `mgr`, `op`, or `cc`.

This command calls the appropriate directive of the **CREATE** primitive to actually create the object. It prompts for and reads in all the necessary information for the invocation of a directive. For the return status see **CREATE(rts)**.

DESTROY(gcp)

Command:

dst - - destroy a node in the Interconnection Schema

Calling Convention:

`dst` *obj*

Description:

This Command prompts for and reads in the name of a node in the interconnection Schema to be destroyed and the name of the cooperation class associated used capability and then calls the **dstnd** directive of the **DESTROY** primitive to destroy the node. The used capability is also destroyed with the destruction of the node, unless it is an outstanding capability over a SendReceive port (see **SEND-RECEIVE(rts)**). This directive requires that the caller has the Destroy-Directory-Node or Destroy-Manager-Node right for its active directory, if a stable capability is used in the destruction of a directory or manager definition node respectively. In any case, the Destroy-Node capcap must be active in the capability.

Obj can take the values `dir` and `mgr`. For the return status see **DESTROY(rts)**.

DROP(gcp)

Command:

drp -- drop a transient capability from c-list

Calling Convention:

drp *obj*

Description:

This primitive prompts for and reads in the name and cooperation class of a transient capability, and then calls the **DROP** primitive to remove the named capability from the caller's c-list. The capability is removed, unless the capability was previously received over a SendReceive port (see **SENDRECEIVE(rts)**), or is a cooperation class capability associated with a port (see **ACCEPT-REQUEST(rts)**). Thus, the capability is removed, only if it is owned by the caller.

Obj specifies the type of the capability to be dropped and can take any of the five values: `dir`, `mgr`, `op`, `cc`, or `pt`.

For the return status see **DROP(rts)**.

HOLD(gcp)

Command:

hld -- make a stable capability, transient

hldc -- create a transient copy of a stable capability

Calling Convention:

`hld` *obj*

`hldc` *obj*

Description:

The `hld` command removes a capability from the caller's active directory and adds it on to the caller's c-list, unless another capability of the same type, name and cooperation class already exists in the c-list. This directive requires that the caller has the Hold right for its active directory and the Hold capcap be active in the capability.

The `hldc` command creates a copy of the stable capability and adds it on to the caller's c-list. This directive requires that the caller has both the Hold and Copy rights for its active directory, and the Hold and Copy capcaps be active in the capability.

Obj can take the values of `dir`, `mgr`, `op`, or `cc`.

This command prompts for and reads in the name and cooperation class of the capability to be hold, and then calls the **HOLD** primitive. For the return status see **HOLD(rts)**.

HOME(gcp)

Command:

home - - change to the login or primary directory

Calling Convention:

```
home
```

Description:

This command changes the caller's active directory to its login or primary directory by calling the **HOME** primitive.

For the return status see **HOME(rts)**.

LOGOUT(gcp)

Command:

log - - terminates the gcp

Calling Convention:

```
log
```

Description:

This command terminates the Gutenberg command processor (*gcp*). *Gcp* calls the procedure *clnenv* of the run-time support to release all the resources that it has allocated, before invoking the C-library procedure *exit*.

MERGE(gcp)

Command:

mrg - - merge a capability with another compatible capability

Calling Convention:

`mrg` *obj*

Description:

This command finds the union of the capcaps and the union of rights of two compatible capabilities and assigns them to the corresponding fields of the first capability. Two capabilities are compatible if they are of the same type, point to the same node of the Interconnection Schema and belong to the same cooperation class.

This primitive requires that the Merge capcap be active in the participating capabilities. If a stable capability is used in the merge, the caller must have, in addition, the Merge right for its active directory.

Obj can take the values of `dir`, `mgr`, `op`, or `cc`.

This command prompts for and reads in the names, cooperation classes and the kinds (*S:* stable or *T:* transient) of the participating capabilities. Then, it calls the **MERGE** primitive. For the return status see **MERGE**(rts).

MODIFY(gcp)

Command:

moda – – modify the active directory capability

modt – – modify the fields of a transient capability

mods – – modify the fields of a stable capability

Calling Convention:

`moda`

`mods` *obj*

`modt` *obj*

Description:

This command modifies the attributes of a capability by calling the **MODIFY** primitive. The command prompts for and reads the name and cooperation class of the capability, as well as the new values of the attributes to be modified. The new values can be the names of the capability and its associate cooperation class, the capcaps or the rights.

The `moda` directive modifies the capcaps and rights of the active directory capability. The `mods` directive modifies a stable capability whereas the `modt` directive modifies a transient capability. The `mods` requires that the caller has the Modify right for its active directory to modify any attribute of the capability. All three directives require that the Modify-Capcap capcap be active in the capability, if the capcaps of the capability are to be modified, and the Modify-Capability capcap be active, if the other attributes of the capability are to be modified.

Obj can take the values of `dir`, `mgr`, `op`, `cc`, or `pt`. For the return status see **MODIFY**(rts).

REGISTER(gcp)

Command:

reg – – make a transient capability stable

regc – – create a stable copy of a transient capability

Calling Convention:

reg *obj*

regc *obj*

Description:

Reg (regc) removes (copies) a capability form the caller's c-list and stores it in its active directory, unless another capability of the same type, name and cooperation class has been registered already.

Both directives prompt for and read in the name and cooperation class of the capability to be registered. Then, the **REGISTER** primitive is called. *Obj* can take the values of **dir**, **mgr**, op, or cc.

This command requires that the caller has the Register right for its active directory, and the capability to be registered must have its Register capcap active. The **regc** directive requires in addition that the Copy capcap be active in the capability.

For the return status see **REGISTER(rts)**.

REMOVE(gcp)

Command:

rmv - - remove a stable capability from the Interconnection Schema

Calling Convention:

rmv *obj*

Description:

This command prompts for and reads in the name and cooperation class of a capability and then calls the **REMOVE** primitive to remove the capability from the caller's active directory. *Obj* specifies the type of the capability to be removed and can take the values of **dir**, **mgr**, op, or cc.

In the case of directory, manager or operation capability, the corresponding Interconnection Schema node is destroyed if the dropped capability is the last one linked to that node. The caller is required to have the Remove right for its active directory while the capability has to have the Remove capcap active.

For the return status see **REMOVE(rts)**.

SCRIPT(gcp)

Command:

script – – execute a command file

quit – – terminate a command file

Calling Convention:

`script`

`quit`

Description:

This command prompts for and reads in the name of a command file, called *script*, to be executed. A script is a collection of gcp commands ended by **quit**. **Quit** terminates the execution of the script and redirects the input of the gcp back to the terminal. If **quit** is omitted, gcp interprets the *end-of-file* marker as **quit**.

Any line in a script starting with % or # is considered a comment and ignored. Each command and its arguments are expected to be on a different line and in the same order as the one prompted by the gcp while reading from a terminal. Although, the notion of nested scripts is not supported, the **script** command can be part of a script. In such case the currently executed script is terminated and execution continues with the first command of the new script.

Gcp, to facilitate the creation of scripts, maintains a history or log file for each session recording all input read. The name of a history file is the concatanation of the user's name, the word *log* and a timestamp, separated by periods. The history files are stored in the *exc* directory of the Gutenberg system (*um2.0/exc*).

The script command searches for a file with given name in the *scr* directory of the Gutenberg system (*um2.0/scr*).

For an example of a script see appendix C.

VIEW(gcp)

Command:

vwact – – view active directory

vwclst – – view c-list

vwcp – – view a capability

vwnd – – view a node of the Interconnection Schema

Calling Convention:

`vwact`

`vwclst`

`vwcp` *obj*

`vwnd` *obj*

Description:

The vwact and vwclst display the caller's active directory and c-list. The vwnd command displays an is-node of *obj* type (dir or mgr). In the case of a directory, a list of all stored capabilities is displayed, whereas in the case of manager definition, all the manager's attributes – – initial active directory, instantiation protocol, associate cooperation class, and a list of all the exported operations, are displayed.

All three directives support partial views. For vwact, vwclst, and vwnd dir, only capabilities of a specific type are displayed. For vwnd mgr, the attributes of a specific operation are displayed.

These directives require that the caller has the View-Node right for its active directory while the View-node capcap must be active in the capability used.

The vwcp directive displays the attributes of a capability: local name, associate cooperation class, capcaps and rights. In the case of an operation capability the attributes include the specifications of the supported operation. It requires that the caller has the View-Capability right for its active directory, if a stable capability is to be viewed. The View-Capability capcap must be active in the viewed capability. *Obj* can take the values of dir, mgr, op, cc, or pt.

The command prompts for and reads in the name of the object to be displayed in the cases of the vwcp and vwnd directives. In the cases of the vwnd it also prompts for the view option. This command calls the appropriate directive of the **VIEW** primitive to bring a copy of the object to be displayed in the user's address space. The displayed format of capabilities is:

local-name | *coop-class*: *kind*

where *local-name* is the name of the capability, *coop-class* is the name of the cooperation class that it belongs to and *kind* is a character string designating the kind of the capability. *kind* can take the values: " " for own capabilities, "*" for borrow capabilities, "u" for client-end port capabilities, "s" for server-end port capabilities, and "u*" and "s*" for borrow client-end and server-end port capabilities respectively.

For the return status see **VIEW(rts)**.

3.3 GCP Library

Here is the complete list of the gcp functions and their interface. These are included in the library *gcp.a*. For almost all the functions the default output is the *sdtout* whereas the default input is the file pointed by *fptr* parameter passed to the functions. All functions that prompt for an input, they write that input to a log file pointed by the global variable *logfptr*.

Some of the gcp commands and their functions, for example ***home*** and ***shomedir***, can be used only by privileged users that have usr directory as their login directory. For this reason ordinary users are advised not to use them although they are included in the *gcp.a*. These privileged commands and functions are not listed below.

```
int crtfls (usrname)
char *usrname;    /* in -- pointer to a character string */

 !
 !  This creates a log file and sets the global variable logfptr. It
 !  returns 0 if success, else -1 for error. The name of the file is
 !  generated by concatanating the supplied usrname with the word
 !  ".log." followed by a timestamp.
 !


void  decode (primtv, object, pcode, otype)
lname primtv;     /* in  -- command to be recognised and encoded */
lname object;     /* in  -- object to be recognised and encoded  */
int *pcode;       /* out -- return command code   */
char *otype;      /* out -- return object code    */

 !
 !  This recognises a gcp command pointed by "primtv" and the object
 !  type pointed by "object". It returns the code for the recognized
 !  command and object in "pcode" and "otype" respectively.
 !


void  execute  (pcode, otype)
```

```
int pcode;      /* in -- command code */
char otype;     /* in -- object code  */

!
! This dispatches the appropriate function for servicing the command
! with code pcode object type otype.
!

extcmd (opname, opcoop)
lname opname; /* in -- operation name */
lname opcoop; /* in -- name of cooperation class */

! This requests the external command opname associated with the opcoop
! coop-class over a SendReceive port.
!
! It invokes the vwcp primitive on the opname operation cap to get the
! details of the operation. Then it prompts for and reads in all the
! arguments. Afterwards it creates a port and invokes sndrcv on the port
! to send the details of the operation and receive the results. Finally
! the port is destroyed by calling dstpt.
!


void gcp (name)
lname name; /* user login name */

!
! This is the control loop of the gcp. It prompts for a new command by
! invoking getcmd, recognises the command by calling decode and services
! the command by calling execute extcmd. It ignores commented input lines.
!
! Initially, it creates the log file by calling crtfls and sets the ctrl-c
! interrupt handler to cancel a command and return control to the gcp main
! control loop.
!
```

```
void  getcmd (primtv, object)
char *primtv;        /* out -- entered command to be executed */
char *object;        /* out -- object on which the command will run against */
```

```
!
!  This prompts for and reads in a gcp command from the file pointed by
!  the global variable infptr. It returns the command and the object in
!  primtv and object variables.
!
```

```
unsigned short  getcpcp (fptr, ctype, capcaps)
FILE *fptr;                  /* in -- input file pointer */
char ctype;                  /* in -- cap type: dir, mgr, op, cc or pt */
unsigned short *capcaps;     /* out -- capcap word (bitmap) */
```

```
!
!  This prompts for and reads in the capcaps of a capbility. It prompts only
!  for the capcaps that make sense for a given capability type one at a time.
!  It returns the read capcap word.
!
```

```
void  getdspecs (bf, noprm, nocap, size)
DT_NODE *bf;             /* in -- pointer to data descriptor buffer   */
int noprm;               /* in -- number of parameters to prompt for  */
int nocap;               /* in -- number of capabilities among the parameters */
int *size;               /* out -- size of the data descriptor buffer */
```

```
!
!  This prompts for and reads in the data descriptors of an operation. First,
!  it prompts for the privileged data and then for the non-privileged.
!  It returns the data descriptor in the passed buffer bf and the size of
!  the data descriptors.
!
```

```
char *getmtmpl (outsize)
int *outsize;  /* out -- size of the operation descriptors buffer */

!
!  This prompts for and fills in a template of a manager to be created.
!  The template contains the manager's general attributes followed by the
!  specs of the exported operations. It returns the allocated manager
!  template and its size.
!
```

```
char *getospecs (size)
int *size;   /* out -- size of the operation descriptor */

!
!  This prompts for the specs of an operation. It allocates an operation
!  descriptor and returns it along with its size.
!
```

```
void getpmodc (prmbuf, capid)
struct prmmod *prmbuf;     /* in -- parameter buffer */
CPSPECS *capid;            /* in -- capability specs */

!
!  This prompts for and gets the new values of the attributes of a
!  capability to be modified. The parameters to be modified and their
!  values are stored in the passed structure of type prmbuf.
!
```

```
struct prmvwnd *getpvwnd (ctype, outsize)
char ctype;       /* in -- type of the node: directory or manager */
int *outsize;     /* out -- size of the allocated vwnd parameter structure */
```

```
!
! This prompts for and reads in the view option (partial or total view).
! It allocates a prmvwnd structure to store the parameters read and
! returns it along with its size.
!


unsigned short  getrights (fptr, rights)
FILE *fptr;                /* in -- input file pointer    */
unsigned short *rights;    /* out -- capcap word (bitmap) */

!
! This prompts for and reads in the rights in a directory capbility
! one at a time. It returns the read right word.
!


void  getword (word, dbuf, idx, wdsize)
char *word;      /* out -- extracted word */
char *dbuf;      /* in -- input character buffer */
int *idx;        /* in/out -- cursor position in the input buffer */
int wdsize;      /* in -- size of the word buffer */

!
! This extracts a word of size wdsize from a character buffer starting
! at the position idx. It ignores leading spaces. It returns the extracted
! word and updates the coursor idx.
!


void logdspecs (dd);
char *dd;   /* in -- pointer to the input data descriptor */

!
! This writes a data descriptor to the file pointed to by the
```

```
! global variable logfptr.
!


void  logmtmpl (mtmpl);
char *mtmpl;   /* in -- pointer to the input manager template */

!
! This writes a template of a manager to the file pointed to by the
! global variable logfptr.
!



void logospecs (opd);
char *opd;   /* in -- pointer to the operation descriptor */

!
! This writes an operation descriptor to the file pointed to by the
! global variable logfptr.
!
!



void  logpcmp (prmbuf);
struct prmcmp *prmbuf;   /* in -- pointer to the parameters of compare */

!
! This writes the specs of the participating caps to be compared,
! their kind (stable or transient) and the kind of comparison to
! the file pointed to by the global variable logfptr.
!



void logpmodc (pbuf);
struct prmmod *pbuf;  /* in -- pointer to the modify param buffer */


!
```

```
!  This writes the new values of the attributes of a capability to be
!   modify to the file pointed to by the global variable logfptr.
!


void  logpmrg (prmbuf);
struct prmmrg *prmbuf;    /* in -- pointer to the parameters of a merge */


!
!  This writes the specs of the participating caps in the merge to the
!  file pointed to by the global variable logfptr.
!


void logpvwnd (ctype, pbuf);
char ctype;                    /* in -- type of the node: dir or mgr */
struct prmvwnd *pbuf;          /* in -- ptr to the parameters of the view */


!
!  This writes  with the view option to the file pointed to by the
!  global variable logfptr.
!


void  printcap (fptr, dbuf);
FILE *fptr;           /* in -- pointer to output file     */
char *dbuf;           /* in -- pointer to the capability buffer */


!
!  This writes a copy of capability returned by the vwcp (view) primitive
!  to the file pointed to by the fptr. Dbuf points to the buffer holding the
!  copied capability.
!


void  printdir (fptr, dbuf);
```

```
FILE *fptr;           /* in -- pointer to output file          */
char *dbuf;           /* in -- pointer to the dir/clist buffer */

!
! This writes a copy of a directory or c-list returned by the vwact,
! vwclst or vwnd primitive to the file pointed to by the fptr.
! Dbuf points to the buffer holding the copied directory or c-list.
!


void printmgr (fptr, dbuf)
FILE *fptr;           /* in -- pointer to output file             */
char *dbuf;           /* in -- pointer to the copied manager node */

!
! This writes a copy of a manager definition node returned by the vwnd
! primitive to the file pointed to by the fptr. Dbuf points to the buffer
! holding the copied manager definiton node.
!


void printopd (fptr, dbuf)
FILE *fptr;           /* in -- pointer to output file  */
char *dbuf;           /* in -- pointer to a copy of an  operation descriptor */

! This writes the description of an operation which is a part of the
! returned buffer by the vwcp or vwnd primitive, to the file pointed
! by the fptr. Dbuf points to the buffer holding the operation descriptor.
!


void quit ();

! This closes the file pointed to by the global variable logfptr and
! redirects the input to the terminal (stdio).
```

```
void sassoc (captype, stable)
char captype;                /* in -- type of the caps: dir, mgr, or op */
boolean stable;      /* in -- kind of cap is stable if TRUE */

!
!  This associates a directory, manager or operation capability with
!  a coop-class by calling the assoc primitive. It prompts for and
!  reads the name and coop-class of the capabilities to be associated.
!

void schgdir ();

!
!  This prompts for and reads in the name of the directory which should
!  become the new active dir, and the name of the coop-class associated
!  with the directory capability. Then it calls the chgdir primitive.
!

void scmpcp (captype)
char captype;                /* in -- cap type: dir. mgr, op, cc or pt */

!
!  This compares two capabilities. It prompts for and reads in the
!  participating caps and the comparison kind. Then it calls the
!  cmpcp primitive.
!

void scopy (captype, stable)
char captype;        /* in -- type of the caps: dir, mgr, op, or cc */
boolean stable;      /* in -- kind of cap is stable if TRUE */

!
```

```
!  This copies a stable (transient) capability and stores it in the
!  caller's active directory (c-list). It prompts for and reads in
!  the names and coop-classes of the old and new caps. Then, it calls
!  the copy primitive.
!


void script ();

!
!  This prompts for and reads in the name of a command file to be executed.
!  It looks for the file in the um2.0/scr directory in order to open it.
!  It also prompts for the execution mode of the script: continuous mode or
!  step mode. In case of continuous mode, it prompts for the period (in sec)
!  between executing each command in the script. It updates the global
!  variable logfptr to point to the newly opened file.
!


void scrtcc ();

void scrtdir ();

void scrtmgr ();

void scrtop ();

!
!  These functions prompt for and read in the necessary parameters to
!  create a coop-class, a directory, a manager or an operation capability
!  by calling the appropriate create primitive.
!


void sdrpcp (captype)
char captype;          /* in -- type of the cap to be dropped */
```

```
!
! This prompts for and reads in the name of a transient capability
! of the type defined by the captype variable. In case of a port
! capability, it calls the dstpt primitive to remove the named
! capability from the caller's c-list. Otherwise, it calls the drpcp
! primitive. Captype can take the values: DIR, MGR, OP, CC, or PT.
!


void sdstnd (ndtype)
char ndtype;            /* in -- the type of the node to be destroyed */

!
! This prompts for and reads in the names of a node of the ICS to
! be destroyed and its associated coop-class. Then, it calls dstnd
! primitive to destroy the is-node. Ndtype designates the type of
! the is-node: dir or manager.
!


void shldcp (captype, copy)
char captype;               /* in -- type of the cap to be held */
boolean copy;               /* in -- copy the cap flag */

!
! This holds a capability by calling the hldcp primitive. It prompts
! for and reads in the names of the capability to be held and its
! associated coop-class. Captype designates the type of cap: DIR,
! MGR, OP or CC. Copy designates whether a copy of the capability
! should be stored in the c-list.
!


void smodact ();
```

```
!
! This modifies the capcaps and rights of the active directory capability
! by calling the the modcp (MODIFY primitive). It calls getcpcp and
! getrights to prompt for the new values of capcaps and rights, respectively.
!


void smodcp (captype, ckind)
char captype;        /* in  -- cap type: dir, mgr, op, cc, pts */
char ckind;          /* in  -- cap kind : stable or transient   */

!
!  This modifies the attributes of a capability by calling the modcp
!  (MODIFY) primitive. It prompts for and reads in the name and coop
!  class of the capability, as well as the new values of the
!  attributes to be modified. The new values can be the names of the
!  capability and its associated coop class, the capcaps or the rights.
!


void smrgcp (captype)
char captype;      /* in -- captype: dir, mgr, op, or cc */

!
!  This merges two compatible capabilities by calling the mrgcp
!  primitive. It prompts for the specs of the particiapting
!  capabilities. The resulting cap replaces the first  of the
!  particiapting capabilities.
!


void sregcp (captype, copy)
char captype;             /* in -- type of the cap to be registered */
boolean copy;             /* in -- copy the cap flag */
```

```
!
! This registers a capability by calling the regcp primitive.
! It prompts for and reads in the names of the capability to
! be registered and its associated coop-class. Captype designates
! the type of cap: DIR, MGR, OP or CC. Copy designates whether a
! copy of the capability should be registered.
!


void srmvcp (captype)
char captype;     /* in -- captype: dir, mgr, op or cc */

!
! This removes a cap from an active directory by calling the
! rmvcp primitive. It prompt for and reads in the specs of the
! capability to be removed. Captype designates the type of
! capability: DIR, MGR, OP or CC.
!


void svwact ();

!
! This displays the active directory of a user. It invokes
! the vwact primitive to bring a copy of the directory in the
! user's address space and then it calls printdir to output it.


void svwclst ();

!
! This displays the c-list of a user. It invokes the vwclst
! primitive to bring a copy of the c-list in the user's
! address space and then it calls printdir to output it.
!
```

```
void svwcp (captype)
char captype;    /* in -- capability type: DIR. MGR, OP, CC or PT */

!
! This displays a capability of type captype. It prompts for
! the specs of the capability and invokes the vwcp primitive
! to bring a copy of the capability in the user's address
! space. Then it calls printcap to output the cap.
!


void svwnd (ndtype)
char ndtype;       /* in -- node type; DIR or MGR */


! This displays a directory or a manager is-node. It prompts for
! the name of the node and the view option and invokes the vwnd
! primitive to bring a copy of the node in the user's address
! space. Then it calls printdir or printmgr to output the node.
!
```

4 Programming Gutenberg Managers

4.1 Introduction

Any application running in the object oriented Gutenberg environment is structured as a set of cooperating object managers. Building an application is a three-step process: the first step involves the design of the managers and the specification of their export operations. The second step involves the system notification of the new managers (new services) by creating new *manager definition* nodes in the Interconnection Schema. This also involves the creation of their execution environment. The last step involves the writing of the actual code with the appropriate Gutenberg calls.

The first step is common to any programming environment and is especially important in an object oriented context. This chapter discusses the last two steps mentioned above. Although the Gutenberg design does not place any restrictions on the programming language used in coding the managers, the current implementation of Gutenberg prototype kernel supports only managers written in C.

4.2 Manager Definitions

Each Gutenberg manager is implemented by a process. The kernel instantiates, as a result of a port creation, a manager using its manager definition in the Interconnection Schema. The operation capability used in the creation of the port points to the appropriate manager definition. The manager definition contains the necessary information for instantiating the manager process such as a cooperation class capability for the file containing the executable image (object module) of the process. This information also includes a directory capability for the *initial active directory* of all manager processes instantiated from the manager definition; the *operation descriptors,* the specification of the operations that are implemented by the manager; the *lifetype* or *instantiation protocol,* indicating how manager processes are instantiated; and, the *interactive* flag indicating whether the manager process is interactive in which case the kernel has to allocate to the manager an input/output device -- *xterm* in the current implementation (the interactive flag is a feature peculiar to the prototype kernel, and would not be a feature of standard manager definitions).

All the capabilities needed by a manager for creating ports to other managers in the same application, or for navigating the Interconnection Schema, should be stored in the manager's initial active directory.

The kernel determines the manner in which ports are connected to the manager processes instantiated from the definition, from the lifetype of the manager and the cooperation class used at the port creation time. The lifetype specifies whether all the object instances of a type are managed by one process or whether each object is managed by different processes. There are three different lifetypes in Gutenberg: *conservative*, *creative* and *class conservative*. In the case of conservative lifetype, a manager process is instantiated from the manager definition only if there is no other manager process executing in the system which was instantiated using this manager definition. If such a process already exists, the port being created is attached to this process. This lifetype provides the means to produce a manager process that manages all the objects of a type,

and to automatically connect port-creating processes to this manager. Using this lifetype, the manager can be informed of the object being accessed at port-creation time using a cooperation class capability. Instantiating more than one conservative manager requires creating more than one manager definition.

Creative lifetype, on the other extreme, creates a new manager process from the creative manager definition is-node for each new port created. This is useful in cases of shortlived managers owned, in effect, by the caller. That is, it allows a process to isolate the newly created manager in order to ensure that the manager cannot leak information. However, it cannot support multi-port interconnections between a specific client and a specific server.

Class concervative allows new managers to be instantiated selectively based on the cooperation class capability supplied at port creation time. The class conservative manager is typically designed to manage one object of the type, and may serve multiple ports from any number of processes.

It is clear that the second step of the development of a gutenberg manager is actually the mapping of the first step, i.e. the design of the manager, to an Abstract Data Type definition (manager definition) in the Interconnection Schema.

A manager definition is created by a process invoking the `crtmgr` primitive (see chapter 2, **CREATE(rts)**). No special privilege is required to invoke this primitive. Upon successful execution of the `crtmgr` primitive, the kernel places a manager definition capability, pointing at the new manager definition and containing its name, in the process's c-list. After the manager definition and the corresponding manager definition capability are created, the process may use the manager capability to invoke the `crtop` primitive to create the operation capabilities for the type. These operation capabilities can then be stored in the Interconnection Schema or distributed over ports to processes wishing to use the type.

4.3 Manager Program Structure

Any manager that executes within the currently supported Gutenberg environment, is expected to be written in the C programming language, and compiled and linked using UNIX commands.

The kernel uses the command-line arguments, *argc* and *argv*, of a manager's program to pass to the manager process the initial values of a number of global variables used in the interface between the kernel and the manager process. The initialization is done by the *setenv* procedure provided by the *run-time support* (see chapter 2). *Setenv* also allocates all the resources needed by the run-time support. The *setenv* procedure must be called before the invocation of any Gutenberg primitive. Its arguments are the command-line arguments:

```
setenv (argc, argv);
```

In addition, to the *setenv* procedure, every manager is required to call the *clnenv* procedure before calling the C-library procedure *exit* or before the end of the manager program. *Clnenv* releases the resources allocated by *setenv* which are not released by the *exit* procedure. Any Gutenberg primitive invoked after calling *clnenv* will fail because of lack of resources, and will

possibly cause the crash of the manager. *Clnenv* does not expect any input parameters and it is also part of the run-time support.

Here is a skeleton program of any Gutenberg manager:

```
main (argc, argv)
int argc;
char **argv;
{

     setenv (argc,argv);
     ...
     clnenv ();

}
```

A typical Gutenberg manager is not expected to use any UNIX signal, except for *cntrl-C* signal (SIGINT). Also, managers are forbidden from using signals 15 (SIGQUIT), 29 and 30 (USER-DEFINED) because they have a special meaning for the run-time support and are bound by it accordingly.

Managers should include the *"shell.h"*, if they use any of the constants, structures or types defined by the system, as, for example, *lname* the capabilities' local name type, or *LNSIZE* the size of the *lname* type.

Sdl.a (software development library) is created to include all the routines needed for developing a manager. It contains all the run-time-support routines, their supporting functions as well as the Gutenberg specific i/o routines.

Interactive managers can be built using some or all of the functions of the gcp (see **gcp-interface(gcp)**). The gcp functions are contained in the library *gcp.a*.

All the include files needed during the compilation of a manager exist in the *incl* directory of the Gutenberg system (*um2.0/incl*), whereas the libraries needed during linking of the manager reside in the *lib* directory (*um2.0/lib*).

The managers' source files are stored in the *svr* directory of the Gutenberg system (*um2.0/svr*). For example, the source files of the file manager are stored in the directory *um2.0/svr/filemgr*. The kernel searches for the image file named by the capability in the manager definition in the *bin* directory of the Gutenberg system (*um2.0/bin*).

In order to automate the compilation and linking of the managers, a skeleton *makefile* is created in the *svr* directory. For each developed manager the makefile should be copied into that manager's directory and updated to reflect any new dependencies. Then the manager can be compiled and linked using the Unix *make* utility.

An example of Gutenberg manager is given in appendix A. The actual code of the Error Manager, presented in appendix A, is shown in appendix B, and the script (see **SCRIPT(gcp)**) to create its manager definition and its environment is shown in appendix C.

5 Implementation Details

5.1 Introduction

This chapter is addressed to those who want to modify or expand the Gutenberg prototype kernel. The goal, in this chapter, is to explain the implementation details and the internal mechanism of the kernel. All major data structures, various modules and their interactions are described in detail so that one can easily gain a global understanding of the system.

The chapter is organized as follows. In the second section, the general organization of the system is described. In the third section, all the important data structures used within the kernel are explained. In the following section, each of the modules is described with all the important functions present in a module. Finally, the life-cycle of a Gutenberg process is illustrated by means of an example.

The typing conventions followed for this chapter are: **boldface** is used for naming structures and functions, `typewriter` font for the fields within structures and *italics* for emphasis.

5.2 System Organization

The Gutenberg prototype kernel is implemented on top of the Ultrix operating system and is written entirely in C. All processes, including the kernel, are Ultrix processes. These processes, conforming to the philosophy of Gutenberg, do not share address space. Communication between the kernel and other processes is done through mailbox facility. The mailbox facility is implemented by using the shared memory mechanism provided by the underlying Ultrix. The Gutenberg system is organized as shown in Figure 5.1. The four primary modules are, *Kernel Control Manager, Capability Directory Manager (cd-manager), Port Manager* and *Process Manager*.

The kernel control manager consists of the critical components of the kernel. They are interrupt handlers, kernel setup, checkpointing module, monitor facility, and mailbox module. The process manager takes care of creation and destruction of processes. The capability directory manager provides the primitives to manipulate the *capability directory* which is also called *Interconnection Schema (ICS)*. It also has a set of basic functions that manage the capabilities. These are extensively used by other managers in the system. Finally, the port manager provides the primitives for the port based operations.

In order to understand the control flow, here is a brief description of how the kernel works. When the system is brought up, the kernel setup calls the checkpoint module to read the stable capabilities from the checkpointed files and reconstructs the Interconnection Schema. It also calls the process manager to create the login processes. After initializing the global data structures and setting up the interrupt handlers, the kernel blocks waiting for an interrupt to arrive.

The servicing of a request is as follows. The requesting process puts the request parameters in the mailbox and signals the kernel. On receiving a signal, the corresponding interrupt handler takes control. The interrupt handler polls the mailboxes of all the processes in round-robin fashion. If there is a new message in a mailbox, the request parameters are read and passed onto the dispatcher. The dispatcher directs the request to a proper primitive-function in capability directory manager or port manager module. The primitive-function in either module passes the results to the mailbox

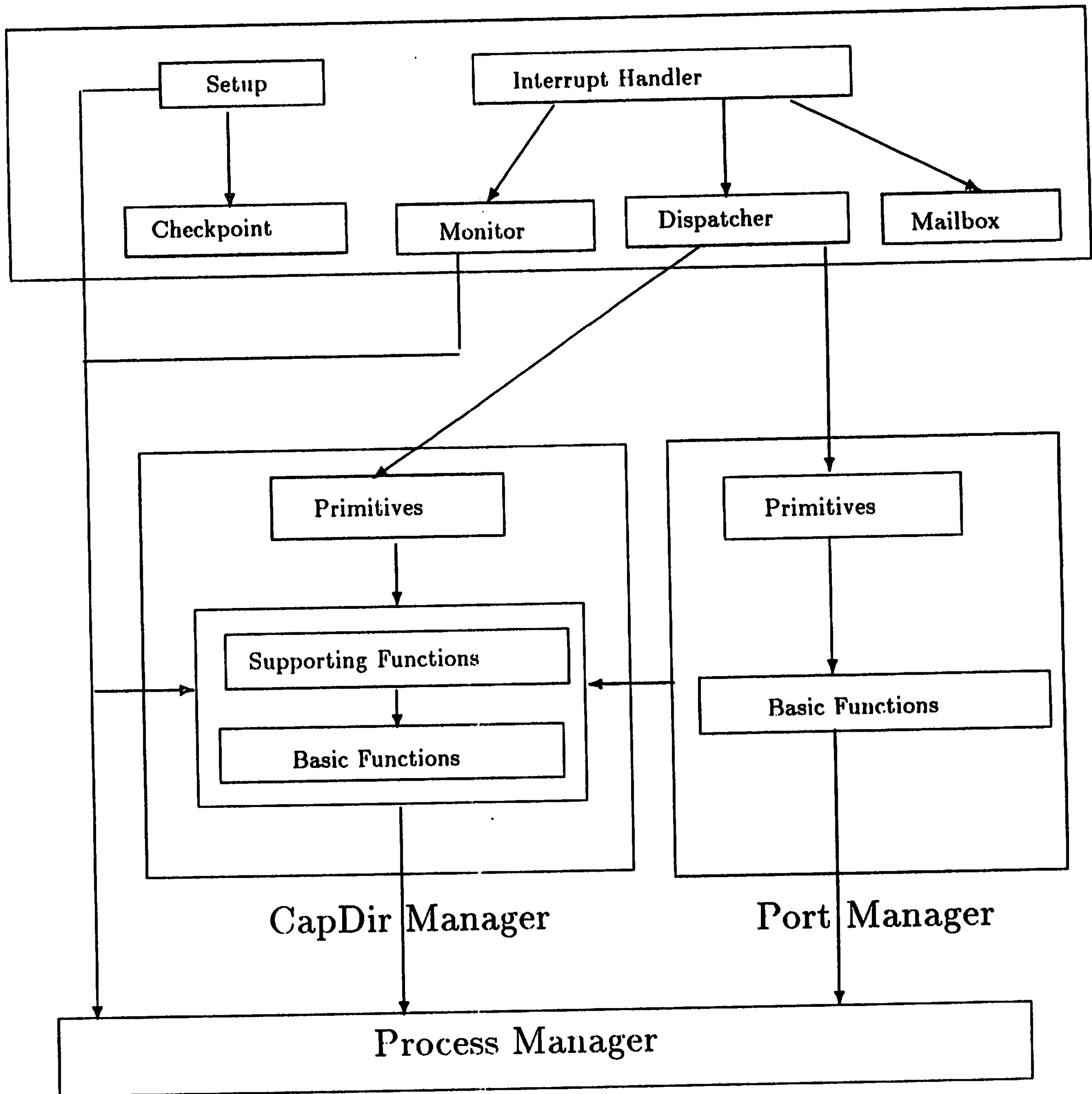


Figure 5.1: Gutenberg System Organization

module which in turn sends them to the requesting process.

When a ctrl-c interrupt is trapped by the kernel, control goes to the monitor. From the monitor, one can traverse through the ICS, take a checkpoint of the ICS, or bring down the system.

5.3 Data Structures

The four major structures in the kernel are, capability directory, process control block, channel control block and mailbox. A good grasp of these data structures is necessary for delving into the various modules that constitute the kernel.

5.3.1 Capability Directory

A listing of the include file **cdir.h** that contains the declarations for all the structures pertaining to the Interconnection Schema and the c-list is given in Figure 5.2. In the following paragraphs, each of those data structures is described in detail. For rest of the chapter, is-nodes and cd-nodes are used synonymously.

There are three kinds of cd-nodes maintained in the system, namely, directory, manager and coop-center. To treat them in a uniform fashion, another structure called **object** is introduced. It houses all three different cd-node structures. The term **object** is not to be confused with the objects that are sent over ports, or the process objects. `Id` is the system-wide unique identifier for the **object**. It is assigned at the time of creation of the cd-node and its value is a function of the current time. The `type` indicates the type of the cd-node (dir, mgr or coop), the **object** points to. The next two fields `lock` and `visited` are used at the time of checkpointing the Interconnection Schema. Depending on the `type` of the **object**, one of the three fields in the union structure will point to directory, manager or coop-center cd-nodes. All the objects in the system are kept in a doubly linked list.

The contents of the three kinds of cd-nodes are described below. One should remember that only a union field in the above **object** structure points to a cd-node. All the capabilities referring to a cd-node actually point to the **object**, and not to the cd-node itself.

The structural cd-node in the ICS is the directory cd-node represented by the structure **subdir**. The directory node consists of a list of capabilities and some house-keeping information. The later information is described first. The `active` field in the **subdir** structure indicates whether the cd-node is a valid node or not. It is set to true on creation and set to false when a process invokes the destroy node primitive. The cd-node is not actually destroyed until all the capabilities pointing to the **subdir** object are deleted. `Use count` gives the number of capabilities pointing to the subdir object. `Active_count` is the number of processes that have the directory as their active directory. And `stbl` points to an array of lists of stable capabilities. There is one list for each type of stable capability, namely dir, mgr, opn and coop. All capability lists in Gutenberg are maintained as doubly linked lists, with each list ordered by the names of the capabilities.

The operational cd-node in the ICS is the manager definition cd-node, **mgrdfn**. It contains, along with some house-keeping information, the list of operations the manager supports, and the

```
/* data descriptor for each argument to an operation.                */

struct dt_node                  /* data descriptor node in an obj   */
{
    int size;                   /* lnth of argument in bytes        */
    char kind;                  /* data or captype                  */
};


struct optype_dscrptr           /* operation descriptor node        */
{
    lname genname;              /* user defined generic name of opn */
    char lnktype;               /* send, receive, send-receive      */
    int objprms;                /* number of arguments in object    */
    int objpriv;                /* number of caps in object         */
    int objsize;                /* total size of the object in bytes*/
    struct dt_node *objdt;      /* data description in the object   */
                                /* it points to an array of dt-node */
    /* below two flds (details) are valid only for sr links         */
    int dtlprms;                /* number of arguments in details   */
    int dtlpriv;                /* number of caps in details        */
    int dtlsize;                /* total size of the object in bytes*/
    struct dt_node *dtldt;      /* data description in rqst-details */
                                /* it points to an array of dt_node */

    struct optype_dscrptr *next;
    struct optype_dscrptr *prev;
};


/* the structure of the object is given below. all three kinds of    */
/* objects are stored under the same structure.                      */

struct object
{
    int id;                     /* system-wide unique id of the obj */
    char type;                  /* object type is dir, mgr, or cc   */
    int lock;                   /* no lock, read or write lock      */
    boolean visited ;           /* for checkpointing                */

    struct object *next;        /* ptr to next obj in the list      */
    struct object *prev;        /* ptr to prev obj in the list      */

    union {
       struct subdir *sdp;      /* ptr to subdir cdnode             */
       struct mgrdfn *mgp;      /* ptr to mgrdfn cdnode             */
       struct coopctr *ccp;     /* ptr to coopctr cdnode            */
    } obj;
};
```

Figure 5.2: Capability Directory Structures (cont.)

```
/* cd-node definitions are given below.                                 */

struct subdir                           /* subdir cd-node                */
{
        boolean active;                 /* flag for garbage collection   */
        int use_count;                  /* no of caps pointing to this node */
        int active_count;               /* active dir for so many users  */
        CAP *stbl[NUM_STBLSTS];
};


struct mgrdfn                           /* manager definition cd-node    */
{
        boolean active;                 /* flag for garbage collection   */
        int mgc_count;                  /* no of caps pointing to this node */
        int opc_count;                  /* no of operation caps linked   */
        int ptc_count;                  /* no of ccbs pointing to this node */
        struct cap *image;              /* ccid for image file           */
        boolean depend;                 /* if true, destroy mgr process when*/
                                        /* all connected ports are destroyed*/
        char protocol;                  /* creative, conservative, classcons*/
        struct cap *initdir;            /* initial active dir            */
        int numofop;                    /* number of export operations   */
        struct optype_dscrptr *oplst;  /* operation export list         */
};


struct coopctr                          /* class cooperation cd-node     */
{
        int cc_count;                   /* no of caps pointing to this node */
        int acs_count;                  /* no of caps associate to this coop*/

};
```

Figure 5.2: Capability Directory Structures (cont.)

```
/* each capability structure is defined below.                          */

struct subdir_cap                   /* subdir capability                */
{
     struct object *dobj;           /* system wide unique directory     */
     unsigned short rights;         /* one bit for each right           */
                                    /* ON (active), off (inactive)      */
};


struct mgrdfn_cap                   /* manager definition capability    */
{
     struct object *mobj;           /* associated mgr node              */
};


struct coopclass_cap                /* cooperation class capability     */
{
     struct object *cobj;           /* system wide unique id            */
};


struct port_cap                     /* port capability                  */
{
  struct ccb *ptid;                 /* ptr to the ccb of the port       */
  CAP *grpcap;                      /* copy of the associated cc-cap    */
};


struct operation_cap                    /* operation capability         */
{
     lname genname;                 /* op's defined name in mgrdfn node */
     char lnktype;                  /* send, receive, send-receive      */
     int objprms;                   /* number of arguments in object    */
     int objpriv;                   /* number of caps in obj            */
     int objsize;                   /* total size of the object in bytes*/
     struct dt_node *objdt;         /* data description in the obj      */
                                    /* it points to an array of dt_node */

     /* following two flds (detail) are valid only for sr links         */
     int dtlprms;                   /* number of arguments in details   */
     int dtlpriv;                   /* number of caps in details        */
     int dtlsize;                   /* total size of the object in bytes*/
     struct dt_node *dtldt;         /* data description in rqst-details */
                                    /* it points to an array of dt_node */

     struct object *mobj;           /* linked to this mgr               */
};
```

Figure 5.2: Capability Directory Structures (cont.)

```
/* all caps are treated uniformly by defining a union structure.     */.

struct cap
{
     lname name;                  /* local name of the cap              */
     lname coop;                  /* associated coop                    */
     char captype;                /* subdir, mgrdfn, coop, op, port     */
     OBJECT *ccptr;               /* ptr to coop-object                 */
     unsigned short capcaps;      /* this fld not valid for some caps */
     struct cap *next;            /* ptr to next cap                    */
     struct cap *prev;            /* ptr to prev cap                    */

     /*depending on the cap type, only one of the following flds     */
     /* will be part of the cap structure.                            */

     union
     {
       struct subdir_cap sd;
       struct mgrdfn_cap mg;
       struct coopclass_cap cc;
       struct operation_cap op;
       struct port_cap pt;
     } ucp;
};
```

Figure 5.2: Capability Directory Structures

image file name for the manager. The active field in the mgrdfn structure serves the same purpose as it does in the subdir cd-node. Mgc count, opc count and ptc count are, respectively, the number of manager capabilities, operation capabilities, and ports (channel control blocks) pointing to the manager definition object. Image points to the cooperation class capability which contains the name of the image file for the manager process. Depend is true if the manager process requires a tty window (xterm in the current implementation). Protocol specifies the life-type of the manager, whether it is a conservative, creative or class-conservative manager. Initdir points to the initial directory capability for the manager process. This is the active directory of the manager process at the time of its instantiation. Numofop is the number of operations exported by the manager. Oplst is the list of all the optype_dscrptr structures for all the supported operations. See below for their description.

The third type of cd-node is called cooperation class cd-node and is not (directly) visible to the user. It has been introduced for implementation reasons. The coopctr structure has only two fields. Cc_count is the number of coop-class capabilities pointing to the structure. Acs_count is the number of capabilities that are associated by this cooperation class. This structure is removed when both the counters reach zero.

As mentioned above, each manager supports a set of different types of operations. A structure is introduced to contain the operation name, corresponding port type, and, amongst others, description of the parameters involved in the invocation of the operation. The structure optype descrptr describes an operation. Users may create capabilities for a particular operation. Those capabilities can have user-specified names. But the operation has a unique name specified at the time of defining the manager. It is called generic name (Genname). Lnktype is the type of the port created for the operation (Send, Receive or Send-Receive). Objprms gives the number of parameters in the operation, objpriv gives the number of parameters that are privileges (capabilities), and objsize gives the total size of the parameters. Objdt is a pointer to an array of dt node structures each of which describes, as explained below, one parameter. The dtlprms, dtlpriv, dtlsize and dtldt fields are detail parameters and have similar meaning as the above object parameters. In case of Send and receive links, whatever sent and received are called object parameters. In case of Send-Receive links, the parameters sent by the client are called details and the parameters sent by the server are called objects. Details do not apply for Send and Receive links. All the operation types supported by a manager are kept in a doubly linked list.

The dt_node structure describes the contents of each parameter in a message passed over a port as part of the operation invocation. Kind indicates whether the parameter is a capability (privilege) or data (non-privilege). Size gives the size of the parameter in bytes, and it is fixed for a capability parameter. An array of dt node structures describes all the parameters in a message.

There are five different types of capability in the system. Since different types of capabilities have different kinds of attributes, there are different structures for them. But for the sake of treating them in a uniform fashion, another structure cap is introduced to house all kinds of capabilities. The cap structure has the union of different capability structures. Name is the name

of the capability, coop is the associated cooperation capability name, and captype is the type of the capability (directory, manager, operation, cooperation class or port). Ccptr points to the coop-center which is the actual coop-id for the above mentioned coop. In case of cooperation class and port type capabilities there is no associated cooperation capability, meaning that both coop and ccptr are null. Each capability is distinguished by its name, coop and type. Capcaps determine how the capability may be modified and used. Depending on the type of the capability, proper structure within the union field of cap is chosen by the accessing routines. At any time, each capability is part of a list, and next and prev point to the next and previous nodes in the list.

Now, the fields within each type of capability are described. The **subdir cap, mgrdfn cap,** and **coopclass_cap** point to their respective object structures. In addition, **subdir_cap** has rights field which is special to directory capability. The **port cap** has a pointer (ptid) to the channel control block (ccb). Each port has a cooperation class capability associated with it. And it is stored at the server-end capability for the port. (This is not same as the associated cooperation class capability explained above. See the section on port manager for further clarification). CHK! Grpcap points to that capability. This field is null for user-end capability. The **operation cap** structure has all the information about the parameters sent or received as part of an invocation of the operation. This information is copied from the operation type descriptor in the corresponding manager definition. Finally, mobj points to the manager object that supports the operation.

5.3.2 Process Control Block

Every process in the system is manifested by its process control block (**pcb**). It contains all the information that pertains to the process. A listing of the include file **pcb.h** that contains the declarations of relevant structures appears in Figure 5.3.

Unixid is the pid of the process as given by Ultrix. Cshid corresponds to the xterm id. Each process in the Gutenberg is instantiated from a manager definition. Mgrptr points to the manager definition object. Each manager process has an associated cooperation class represented by ccgrp. The process control block is created even before the process is instantiated and may exist beyond the life-time of the process. Status can take different values: when the pcb is created, status is set to instantiated; after the creation of process, it gets set to *created*; when the process notifies the kernel, it is set to *active*; the process goes to a blocked state if it is waiting for some event to occur; and status is set to *terminated* when the process exits or gets killed.

A process may be blocked for a variety of reasons. For example, a server can be made to block until a new port is created. Or a process can invoke it synchronous Send operation and may get blocked until the receiver makes the corresponding Receive operation. Event is the request id on which the process is blocked. Blkd_on specifies the port on which the process is blocked. It is null if the process is a server that is blocked on a new port.

A process always starts with its login directory as the active directory. The login directory is static and defined in the manager definition as the initial directory. The process can change its active directory during its life-time. Logdir and actdir are the respective capabilities for them.

```
/* iname structure contains info about the caps that are intended    */
/* to be sent (bot not yet delivered) or caps that are intended       */
/* to be received (but not yet received).                             */

struct iname {
     lname name;                  /* intented lname for a cap         */
     lname coop;                  /* and its coop class               */
     struct iname *next;
     struct iname *prev;
};

/* process control block structure                                    */

struct pcb {
     int unixid;                  /* ultrix process id                */
     int cshid;                   /* the original cshell              */
     OBJECT *mgrptr;              /* ptr to the mgrdfn object         */
     OBJECT *ccgrp;               /* id of coop-class                 */
     char status;                 /* active, terminated or blocked    */

     CCB *blkd_on;                /* port, process may be blocked on  */
     int event;                   /* rqst# (any event) which  process */
                                  /* may be blocked on                */
     char *args;                  /* arguments of the blocking event  */

     CAP *actdir;                 /* ptr to cap for active directory  */
     CAP *logdir;                 /* ptr to cap for login  directory  */

     CAP *trns[NUM_TRNSLSTS];     /* array of transient cap lists     */
     CAP *unacpt_pts;             /* list of undelivered port caps    */
     struct iname *intent[NUM_INTNTLSTS];
                                  /* array of lists of intent caps    */

     /* following dcls are concerned with shared memory               */
     struct shlmbx *shlmbox;      /* ptr to this process's mailbox    */
     struct knlmbx *knlmbox;      /* ptr to kernel's mailbox          */
     int mbxid;                   /* shared mem id of  mbox           */

     int tty;                     /* id of the assigned tty; -1 no tty*/
     struct rsponsq *tail;        /* tail of the queue of responses   */
     boolean fplock;              /* to control recursion of free-pcb */

     struct pcb *next;
     struct pcb *prev;
};
```

Figure 5.3: Process Control Block Structures

Theses capabilities are copied from the initial directory capability in the manager definition.

The c-list of a process is maintained as a number of transient lists, one list for each type of capability. **Trns** points to the array of those lists. Each element of the array is a circular list of capabilities. Similarly, the intention list is maintained as an array of lists, each element being a doubly linked circular list of inames. See below for further explanation on intention lists and inames.

When a user creates a port, both user and server capabilities for the port are created. The server capability is added to the not yet accepted list of ports **unacpt pts**. When the server accepts the request, the capability is moved from the unaccepted ports list to the transient list.

`Shlmbx` and `Knlmbx` are pointers to the shell and kernel mailbox pair. `Mbxid` is shared memory identifier for the mailbox pair. For further details, see the section that describes mailbox. **Tty** is the window id for the process. **Tail** points to the tail of the list of responses the kernel prepared for the process. They are waiting for the process mailbox to get empty. **Fplock** is used by **freepcb** to control recursion. There are two lists of pcb's, called *active_pcb list* and *zombie_pcb list.* After creating a pcb, it is put in the active pcb list. When the corresponding process is terminated, it is moved to the zombie_pcb list. It stays in the zombie_pcb list until the side effects of termination are taken care of, at which time the pcb is deleted from the zombie pcb list and the existence of the process is totally forgotten by the system. Both lists are doubly linked circular lists, with a dummy header, ordered by the value of the pointer to the manager definition.

The **iname** structure is to store the intended names. The following scenario makes it easy to understand the need for such a structure. Assume that a process sends a capability (say from c-list) over a port and the receiver has not yet received. This capability should be removed from the sender's c-list to prohibit the sender from further sending the same to other receivers. Also, the sender should not be allowed to add a capability, with the same specifications, to the c-list because of the following reason. If the receiver terminates for some reason, the capability is returned to the sender's c-list. And then there are two capabilities with identical characteristics. To avoid these problems, kernel maintains for each process, an intention list of the capabilities that are sent or to be received. Before making any additions to the c-list it is made sure that there are no conflicts with the intention list.

The fields within the iname structure are **name** and **coop** of a capability. And the list is organized as a doubly linked list. The **type** of the capability is not part of the structure because there are different intention lists for different kinds of capabilities.

5.3.3 Channel Control Block

In order to invoke any operation, a process must first create a port to connect itself to the manager that supports the operation. To create a port, the process submits an operation capability and optionally cooperation class capability. For each port created in the system, a channel control block (ccb) is created. It contains all the information that pertains to the port. A listing of the include file **ccb.h** is given in Figure 5.4.

```
/* the definition of the channel control block structure              */

struct ccb {
     lname opname;                  /* generic operation name           */
     OBJECT *optype;                /* manager exporting the operation  */
     char lnktype;                  /* send, receive, send-receive      */
     OBJECT *grpid;                 /* coop class related to port       */

     /* descriptor of object passed on this port                        */
     int objprms;                   /* number of arguments in object    */
     int objpriv;                   /* number of privileges in object   */
     int objsize;                   /* object's size in bytes           */
     DT_NODE *objdt;                /* description of data in  message  */

     /* detail arguments make sense only in case of sr link type.       */
     int dtlprms;                   /* number of arguments in details   */
     int dtlpriv;                   /* number of privileges in details  */
     int dtlsize;                   /* details's size in bytes.         */
     DT_NODE *dtldt;                /* data description in rqst-details */

     PCB *usrid;                    /* pointer to user's pcb.           */
     CAP *usrcap;                   /* points to the user-end port cap  */
     boolean dnglusrcp;             /* is usr-end on a dngling list     */

     PCB *svrid;                    /* ptr to server's pcb.             */
     CAP *svrcap;                   /* points to the server-end port cap*/
     boolean dnglsvrcp;             /* is svr-end on a dngling list     */

     struct ccblog *usrstk;         /* stack of SR-ports over which this*/
                                    /* port's use-privilege was passed. */
     struct ccblog *svrstk;         /* stack of SR-ports over which this*/
                                    /* port's serve-privilegewas passed.*/

     /* each node in the following queues is pendig for something       */
     struct objnode *objq;          /* objs pending to be received.     */
     int noofobj;                   /* no of objects on the object queue*/
     struct objnode *dbufq;         /* buffers to receive an object.    */
     struct objnode *dtlq;          /* buffers to receive req-details.  */

     /* the following three fields apply only for SR ports              */
     struct ostdcap *ostdlst;       /* outstanding caps list on the port*/
     boolean pt_busy;               /* is SR primitive in progress?     */
     boolean has_dtls;              /* did server invoke get-details?   */

     boolean dst_onclr;             /* destroy the port after clearing? */

     struct ccb *next;              /* pointer to the next ccb.         */
     struct ccb *prev;              /* pointer to the prev ccb.         */
};
```

Figure 5.4: Channel Control Block Structures (cont.)

```
/* definitions of structures used within the ccb                    */

struct ccblog                   /* item of the SR port-stack.       */
{
     struct ccb *ptid;          /* ptr to the port's CCB.           */
     struct ccblog *next;       /* pointer to the next item on stk. */
};

struct ostdcap                  /* item of the outstd-cap list.     */
{
     CAP *origcap;              /* ptr to the original cap.         */
     CAP *copycap;              /* ptr to a copy of the cap.        */
     boolean copy;
     char capkind;              /* stable, transnt, uown, ubrw, etc.*/
     struct ostdcap *next;      /* ptr to the next outstanding cap. */
};

struct objnode                  /* item in the previous queues.     */
{
     int msgid;                 /* msg thru which object is passed. */
     boolean ackdue;            /* is acknowledgement due?          */
     char *dbuf;                /* ptr to object's nonprivilege data*/
     struct ostdcap *dnglst;    /* ptr to privilege data.           */
     struct objnode *next;      /* ptr to next object.              */
};
```

Figure 5.4: Channel Control Block Structures

Opname is the generic operation name for which the port is created. Optype points to the manager definition object that supports the operation. Lnktype specifies the directionality of the port - Send, Receive or Send-Receive. The above fields are copied from the operation capability. Each port has an associated cooperation center. At the time of creating a port, the user gives an operation capability, and a cooperation class capability. If the operation does not have associated cooperation class, then a copy of the given cooperation class capability is used. Otherwise, a new capability is created from the operation capability's associated cooperation class capability. In any case, the newly created or the copied capability is used as the associated cooperation class capability for the port. And grpid is a pointer to the cooperation class center associated with the port. This is the coop-center for the given associated cooperation class capability. If the associated cooperation class capability is specified to be null, then the coop-ctr associated with the operation capability is used. The fields that describe the contents of message parameters on the port are copied from the operation capability structure. And they have exactly the same meaning as explained before. The duplication of this information is required because, once the port is created, the process may decide to drop the operation capability. When a port operation is invoked, the duplicated fields come in handy to check the validity of the parameter values.

Usrid (Svrid) points to the process control block of the user (server) of the port. When a new port is created, two port capabilities are created, one for each end. Usrcap (Svrcap) points to the user-end (server-end) capability of the port. Dnglusrcp (dnglsvrcp) is true if the user (server) end is dangling in the sense it is being sent or lent over a port. The flags get reset when the transfer is complete. All the Send-Receive ports over which a port capability is lent are linked into a list called ccblog. This list, organized as a stack, is helpful in sending back the port capability to the original user or sever. Usrstk and svrstk are the two ccblogs for the user and the server ends of the port.

All the messages on the port, pending to be delivered, are kept in a queue called objq. Noofobj is the number of messages on the queue. When the receiving process makes the port request before the sender has posted the message, then the receiver's buffer is put in a queue called dbufq. Dtlq applies only for Send-Receive ports. It is the queue of the details buffers given by the server of a Send-Receive port. Here is an explanation of what these queues hold for the three different types of ports. In case of Send and Receive ports, what the user sends (objects) are queued on objq and what the receiver sends (buffers) are queued on dbufq. In case of Send-Receive ports, the objects and the buffers sent by the user as part of the Send-Receive call are kept in objq and dbufq respectively. And the buffers the server sends as part of the get-details call are kept in dtlq.

All the capabilities lent over a Send-Receive port are called outstanding capabilities. **Ostdlst** points to the list of such capabilities on the current port. **Pt busy** indicates if a Send-Receive primitive is in progress on the port. **Has_dtls** is true if the server has obtained the details (**get_details**) and not yet invoked the **gp send** primitive to complete the primitive. The above fields apply only to Send-receive ports.

Dst_onclr is set to true if the port has to be destroyed once it is made clear of any port-operation

under progress.

As explained above, there is a stack of ccb's indicating the traversal of a port capability. The stack is a list of **ccblog** nodes, each node having a pointer (`ptid`) to the ccb over which the port capability passed over.

Each node in the outstanding capability list has **ostdcap** structure. `Origcap` points to the original capability that is being sent or lent. `Copycap` points to a copy of that capability. It is null, if the capability is an exclusive capability (like a port capability) or is being passed without copying (as indicated by `copy`). `Capkind` specifies the kind of capability – stable, transient, user-end, server-end, owned or borrowed.

All of the queues mentioned above have nodes of similar structure (**objnode**). `Msgid` is the request id of the process. `Ackdue` indicates whether the request requires an acknowledgement. `Dbuf` points to the non-privilege portion of the message. And `dnglst` points to the list of privileges (capabilities) that are part of the message. This list is prepared after validating the request.

5.3.4 Mailbox

Communication between the kernel and any user process (also called shell) is done through mailboxes. And the mailboxes are implemented using the shared-memory facility. There is a mailbox pair, called shell mailbox (**shlmbx**) and kernel mailbox (**knlmbx**), for each user process. A shell posts requests in its kernel mailbox that is read by the kernel. The kernel posts the responses in a shell mailbox that is read by the corresponding shell. Each of these mailboxes have a header structure. The request parameters and the response follow the headers. A listing of the include file **mlbox.h** that contains the mailbox structures is given in Figure 5.5.

`Rqstid` is the identifier given by the shell for the request. The kernel uses the same for posting the response. `Opcode` is the code for the kernel primitive. `Cpname`, `coop` and `kind` are respectively the name, associated cooperation class name and type of the capability the shell used. These fields may have `null` values for some of the primitives. `Ack` specifies if an acknowledgement should be made for the request. `Sync` specifies if the shell process must be blocked until the request is satisfied. `Mbxsize` gives the size of the parameters put in the mailbox. If the mailbox size is not enough to keep the parameters, the shell creates a new shared-memory segment and puts the rest of the parameters in that. The shared memory id and size are given in `shmid` and `shmsize`. `New` indicates whether the message is read or not. The shell sets it to true after posting it and the kernel resets it after reading it.

The fields in the **shlmbx** structure are similar to those in **knlmbx**. `Rqstid` is same as what was given by the shell for the request. This makes it convenient for the shell to match the response with the corresponding request. `Status` is the status of the request. `Eor` is true if the request is completed. `Mbxsize` gives the size of the response contents in the mailbox. If the mailbox size is not enough to accommodate the full response, a shared memory segment is created for rest of the response. Its id size are given by `shmid` and `shmsize`. `New` indicates if the message in the mailbox is new. The kernel sets it to true after posting the response and the shell sets it to false after reading

```
/* the shell mailbox structure                                  */

struct shlmbx {
     int rqstid;         /* shell generated request id          */
     int status;         /* value of returned status            */
     boolean eor;        /* end of request flag                 */
     int shmid;          /* shared memory id holding the response */
     int shmsize;        /* size of shared memory               */
     int mbxsize;        /* size of data in mailbox buffer      */
     boolean new;        /* flag showing unread response        */
};

/* the kernel mailbox structure                                 */

struct knlmbx {
     int rqstid;         /* shell generated request id          */
     short opcode;       /* code of the requested operation     */
     lname cpname;       /* local name of a capability; it is used */
     lname coop;         /* its coop class                      */
     char  kind;         /* its type -- dir, mgr,...            */
     boolean ack;        /* flag for pending ack for port operation*/
     boolean sync;       /* flag for blocking request           */
     int shmid;          /* shared memory id holding the parameters*/
     int shmsize;        /* size of shared memory               */
     int mbxsize;        /* size of data in mailbox buffer      */
     boolean new;        /* flag showing unserviced request     */
};
```

Figure 5.5: Mailbox Structure

it.

5.4 System Modules

In this section, various modules that make up the Gutenberg kernel are described. The four major system components are Kernel Control Manager, Capability Directory Manager, Process Manager, and Port Manager. In the following subsections, the functionality of each component, along with the functions that comprise it, is described in detail.

5.4.1 Kernel Control Manager

Kernel Setup

The function **main** of the Gutenberg system is part of the **setup** module, and is called **kernel**. The kernel calls **knlsetup**. **Knlsetup** binds all the signal handlers, sets up the mailbox facility for interprocess communication, and initializes some global data structures like the pcb lists and the ccb list. There are two different modes in which the kernel can be set up. In the single user mode, the root is created along with the user and the system partitions, by calling **setroot**. This mode is useful for creating a new ICS. In the other mode, the ICS is reconstructed from checkpointed versions by calling **boot**.

Then the **kernel** calls **getusr** in the process manager to create the login processes. **Kernel** then calls **setup tmr**, enters an infinite loop and **pauses** in the loop. **Setup tmr** sets an alarm interval so that the kernel gets an alarm signal at fixed intervals of time. Gutenberg is an interrupt-driven system. Whenever there is a signal to the kernel, the control goes to a proper interrupt handler and from there returns to the loop and pauses until the next interrupt arrives.

Here is a description of some of the routines that are part of the **setup** procedure. **Setconfig** reads the system configuration values from the administrator database files. **Setwtty** reads the xterm parameters from a tty database file and initializes tty structures. **Setpcblsts** creates a doubly linked circular list of pcb's with a dummy header. **Setccblst** creates a list of ccb's. **Setroot** creates the root directory, the user and the system partitions. It is described under the capability directory manager section. **Boot** calls **restore_ics** of checkpoint module to reconstruct the ICS.

Mailbox

The mailbox module provides the interprocess communication facilities through shared memory. At kernel setup time, **crtmbx** is called to allocate a chunk of shared memory. This memory is treated as an array of mailbox pairs, one pair for each process in the system. Each mailbox pair consists of a kernel mailbox and a shell mailbox. The size of the shared memory is chosen to be sufficient to accommodate the maximum number of processes (configuration parameter) in the system. At the time of bringing the system down, **dstrmbx** is called to destroy the mailbox and deallocate the shared memory.

At the time of creation of a process (see **crtprs**), the **newmbx** function is called. It searches for a free mailbox pair and returns the index of the pair in the array. This index is passed on to

the created process. That process uses the index to access its mailbox. When a process terminates, **childterm** calls **freembx** to free the process's mailbox pair. The mailbox pair then becomes available for a future process.

When there is a request from a process, the kernel calls **rdkmbx**. It reads the contents of the kernel mailbox (and any attached shared memory) of the process into kernel's private address space. The attached shared memory is used when the mailbox size is not enough. After reading the attached shared memory segment is released.

Knlrtn is called by each Gutenberg primitive function to return the results of a request to the user process. If the mailbox is empty, **knlrtn** puts the response in the mailbox (**wrtsmbx**), and signals the user process. Otherwise, the response is queued (**enqrspons**) for later delivery. Each process has a queue of the responses to be delivered to the process. **Deqrspons** dequeues a response from the front of the queue. And **fwrdrspons** forwards the response by putting the response in the mailbox and signaling the user process.

Wrtsmbx is called to send the results of a request to a process, it writes the given results into the shell mailbox of the process. If the mailbox size is not enough, an extra shared memory segment is created.

The mailbox module provides the following functions for the user process to communicate with the kernel. These functions are called from the routines in the run-time support package (**rts**). At the time of setting up the environment (**setenv**), the user process calls **openmbx** with the mailbox id provided to the process by the kernel. Openmbx attaches the shared memory segment to the process. Before exiting, the user process cleans up the environment (**clnenv**) and then calls **closmbx** to detach the shared memory (**shmdt**).

Wrtkmbx copies the request parameters into the kernel mailbox. As above, an extra shared memory segment is created and used if the mailbox is not big enough. **Rdsmbx** reads whatever the kernel had posted, into the user provided buffers. If the kernel used extra shared memory, it is released after copying the contents.

Hdkmbx is called by the kernel to find if there is a new message from the user. **Hdsmbx** is called by the user to find if there is a new response from the kernel.

The following shared memory functions are called by the above mailbox implementation. **Shmalloc** allocates required amount of shared memory; **Shmattach** attaches a given shared memory segment id and returns the starting address of the segment for the caller to use. **Shmdstr** destroys a shared memory segment.

Interrupt Handler

Different signals are used in the Gutenberg system to indicate occurrences of different events; SIGML for new mail, SIGALRM for alarm, SIGTERM for termination of a child process, and SIGINT for ctrl-c. Different handlers are bound to different signals at the kernel setup time. The handlers for the above signals are **knlicntrl, knltmr, sigchld** and **ctrl c** respectively.

Knlicntrl handles the SIGML. It polls each process's mailbox. If there is a new request, the mailbox is read and **dsptcher** is called to dispatch the request to the corresponding primitive

supporting function. If there is a response in the mailbox that is not yet read by the process, SIGML is sent to the process. If there is a pending response to be posted in the mailbox, and if the mailbox is now found to be empty, **fwrdrspons** is called to deliver the message.

At the setup time, an alarm is set to go off at periodic intervals. When the alarm goes off, **knltmr** is called to service the signal. **Knltmr** simply sends SIGML to the kernel process. The alarm is required, amongst others, to periodically look into the mailboxes and see if there are new requests to process or new responses to be sent. Since Ultrix does not guarantee the delivery of all signals, the periodic check up is mandatory to account for, say, a loss of SIGML from a user process.

Ultrix delivers the SIGCHLD to the kernel whenever a child process (user process) terminates. **Sigchld** finds the pcb of the terminated process and calls **childterm** to process the termination of the process. **Childterm** is described in the Process Manager section.

Ctrl c hanlder calls **monitor**, which is discussed in the next section.

Monitor

When ctrl-c is pressed at the kernel process window, the signal handler, **ctrl c**, calls **monitor**. The system administrator can use the **monitor** facility to traverse through the ICS, take checkpoints, and for other administrative purposes. **Monitor** provides different ways of bringing the Gutenberg system down. *Abort* option is used to abruptly exit the system. With this option, all processes are immediately terminated, and the kernel exits without taking any checkpoints. *Exit* option terminates all the processes, brings the ICS to a consistent state, and takes a checkpoint optionally. *Shutdown* option is used to give a warm start to the kernel.

Knlabort kills all the processes, deallocates the shared memory used for the mailbox and exits the Gutenberg system. **Knlshutdown** kills all the processes without affecting the consistency of the ICS. **Knlexit** calls **killchld** (process manager) to kill all the user processes. If requested, a checkpoint of the ICS is taken, and the shared memory used for the mailbox is deallocated. And finally, Ultrix system call *exit* is called to terminate the Gutenberg system.

Checkpointing Module

This module has the responsibility of checkpointing ICS to secondary storage for a later retrieval at the time of booting the system. The algorithm employed is a breadth first traversal of a graph with heterogeneous nodes and links. To avoid duplicate checkpointing of nodes `visited` field is set once a node is traversed. Capabilities pointing to inactive nodes are deleted while traversing the graph.

Chkpoint is called by **knlexit** to checkpoint the contents of ICS onto files. Starting from the root directory object, it calls different functions to checkpoint different kinds of objects until all objects are checkpointed. **Ckpoint_dir** is called to checkpoint the directory object. Ckpoint_dir copies the object contents onto the checkpoint file and calls **ckpoint cap** for the four lists of cappabilities present in the directory. **Ckpoint_mgr** copies the manager object contents onto the checkpoint file and calls **chkpoint cap** to checkpoint the login and image capabilities. It also

calls **ckpoint_optype** to copy the operation descriptors. **Ckpoint_cap** takes a list of capabilities and first deletes the capabilities that are pointing to inactive cd-nodes. Then each capability is checkpointed. The object pointed to by the capability is added to the queue of objects for their checkpointing. After checkpointing the entire ICS, **chkpoint** calls **chg_version** to make the most recent checkpointed files as the current version files.

Restore_ics is called by the **boot** function at the kernel setup time. It reads the ICS from the checkpointed files and reconstructs the ICS that existed at the previous checkpoint time. It calls **restore_subdir, restore_mgrdfn, restore_cc** who in turn call **restore_cap** or **restore_optype** functions. All these functions do the reverse of what their corresponding checkpointing functions do.

5.4.2 Capability Directory Manager

In this section, the basic routines that are used for manipulation and maintenance of the underlying data structures concerned with capabilities are described first. These basic functions are used by process manager, port manager and of course capability directory manager. Then the support routines are presented. They are one level above the basic routines and are called by the top level capability directory manager primitives. Finally, the functions that correspond to the Gutenberg Capability Directory Primitives are presented. **Dsptcher** calls these functions to service the requests of user processes.

Basic Functions

Capnew is called to create a new capability and initialise the fields to null. **Capfree** deallocates the space occupied by structures pointed by the fields within a capability, and also frees the **cap** structure itself. **Capenq** adds a given capability to the tail of a capability list. **Capdeq** deletes the first capability from a list of caps. **Capins** inserts a given capability into a capability list at a specified position. **Capadd** adds a capability to a given capability list at an appropriate position to keep the list sorted. **Capcpy** copies a capability contents into a newly created cap. It also increments the reference counter of the corresponding cd-node.

Capsrch is called by most routines, to find if a capability exists in a process protection domain. The caller can specify what capability lists of the process should be searched and also the order of search. A pointer to the found capability is returned. **Capsrch** calls different search routines to do the searching. **Srch_a** searches for a capability in the active directory of the process. **Srch_c** searches in the c-list, **srch_p** in the port list and **srch_i** in the intention list of the process. All of them, except for **srch_i**, invoke **capfnd** to find the capability in a capability list. **Srch_a** and **srch_c** check (if requested), if the object pointed to by the capability is inactive. In that case, the capability is removed from the list and it is as good as not finding it in the first case. **Chk_grbg** is the function that checks if the capability is garbage (points to inactive cd-node) and calls **releasecap** to remove the capability from the list.

There exists an intention list for each process. It contains the specifications of the capabilities that are intended to be sent or received, but whose transfer is still in transition. **Insilst** inserts a

given capability in an intention list at a specified position. **Addiname** adds an intended capability at the appropriate position to keep the list sorted. **Isiname** checks if a given capability is an iname (is it present in the intention list?). **Delilst** deletes a given capability from the list.

Capfnd is a basic routine called by the above search routines (except for **srch_i**), and also by other modules. It checks if a given capability exists in a capability list.

Releasecap deletes a capability from a capability list and frees the capability itself. If the reference count of the object pointed to by the capability becomes zero, the object is also deleted by calling **freeobj**. **Freeobj** frees the object and also calls **releasecap** to free any capabilities present in the object; for example, in case of directory object, it releases all the capabilities present; in case of manager definition object it releases the initial dir capability and the image cap. The cd-node pointed to by the object is also released.

Supporting Functions

The following supporting functions are called by the primitives. **Dirnew** creates a directory cd-node and a directory object to house the cd-node. Similarly, **mgrnew** and **ccnew** create corresponding cd-nodes and objects. **Opnnew** creates an operation type descriptor structure.

Opnadd adds a given operation descriptor to a list of them. Given a name and a descriptor list, **Opnfnd** finds the operation that has the same generic name. **Opnmod** modifies an operation's characteristics. But in practice, it is used to fill up the empty fields of a newly created operation with the actual values. **Opnvld** finds if a given operation is valid (supported by a manager).

When the user wants to change the active directory, a check is made to see if the new directory capability is a valid one. Similarly, before creating a new operation capability, a check is made to see if the manager capability is valid. Here valid means, the user process should have not only the directory or manager capability but also the cooperation class capability associated with it. **Vldcap** does the required check. It checks if the user has the required capability and then checks if the user has a cooperation class capability pointing to the same **coop-center** as the associated cooperation class.

Dirpack packs all the capabilities in a given directory. This is used to view the active directory or any directory cd-node. **Mgrpack** packs the contents of the cd-node for viewing the manager definition cd-node. **Clstpack** packs the contents of the c-list of a process. This is called by the view c-list primitive.

Capability Directory Manager Primitives

In this section, all the primitive-functions are described. To start with, the **create** primitives are as follows. **Gdircrt, gmgrcrt** and **gcccrt** create the three different kinds of cd-nodes. For **gdircrt** and **gcccrt**, only their names are required from the user; but for **gmgrcrt** all the specifications of the supported functions and other details about the manager image file, initial directory etc., are also required. **Gmgrcrt** creates, as part of the cd-node, a list of operations supported by the manager. **Gopcrt** creates an operation cap. The user process supplies the generic name of the operation, and the manager capability. It checks if the user has the manager capability and if the

manager supports the given operation. The specifications of the operation are copied from the mgr cd-node to the newly created operation cap. In all these create primitives, corresponding capabilities are created and stored in the c-list of the calling process. Always a check is made to make sure that there are no capabilities with the same specifications in the c-list.

The directory, manager and operation capabilities have an associated coop-class with them. When the capabilities are created with the above primitives, the association is left unspecified; but this can be changed by means of gassoc. Gassoc disassociates the coop-class (if any) of a capability and assigns a new coop-class (if any). The user must have a capability for this new coop-class. The **acs count** of the previous **coop center** is decremented and that of the current **coop_center** is incremented. A pointer to the coop cd-node always exists in the field, **ccptr**, of the **cap** structure. It is null for unspecified associations. If the coop cd-node's reference counters (**acs_count** and **cc_count**) reach zero, the cd-node is destroyed.

Gdstnd is called to destroy a directory or a manager cd-node. (The coop cd-node is not visible to the user). If the user has the specified cap, and the destroy right, the cd-node is marked inactive and the capability is released. The actual node is destroyed, when all the capabilities pointing to the node are released.

Gregcp is called to register a transient capability in the active directory. The user can specify whether to copy or move the capability from the c-list to the active directory. **Ghldcp** does the reverse. It copies or moves a specified capability from the active directory to the c-list of the user process. In both cases, it is checked to make sure that the destination does not already have a capability with the same specifications.

Gdrpcp is called to drop a specified capability from the c-list. The capability must be owned by the process. (Borrowed capabilities should be returned properly and hence cannot be dropped by a non-owner). **Grmvcp** removes a specified capability from the active directory if it has the proper right. In both cases, if the reference counters to the cd-node pointed to by the released capability reaches zero, then the cd-node is destroyed.

The user may modify some of the attributes of a capability or the active directory. In the case of capability, they are name, associated cooperation class name, capcaps and rights; and for active directory, they are capcaps and rights. **Gmodcp** checks for authorization and conflicts before modifying the attributes.

Gmrgcp merges two capabilities. It finds the union of the rights and the union of the capcaps of two compatible capabilities and assigns them to the corresponding fields of the first of the two capabilities.

Gcmpcp compares two capabilities. User may choose different kinds of comparison: Do the capabilities point to the same is-node? Do they have same cooperation class? or both.

Gdirvwac is called to view the active directory. It calls **dirpack** to pack the contents of the directory into a buffer. A process can call **gdirvwcl** to view its c-list, which in turn calls **clstpack** to pack the contents. **Gvwnd** is called to view a directory or a manager cd-node. It calls **dirpack** or **mgrpack** to pack the contents. In all of these view commands user can specify if the contents

of the object are to be viewed in full or in part. **Gvwcp** packs the contents of a capability for viewing. It calls **cappack** for packing.

One of the most frequently used primitives is *change directory*. **Gdirchg** changes the active directory to a directory the given capability points to. **Gdirhome** changes the active directory to the login directory of the user process.

5.4.3 Process Manager

Process Manager is responsible for managing the pcb's, creation and destruction of processes, and cleaning-up operation after the termination of a process. The only function in this module called by the **disptcher**, as the result of a user process request, is **noteid**, an implementation primitive not visible to the users. Every Gutenberg process must notify the kernel of its **unixid**. This is necessitated by the interactions of the kernel and the Xwindow facility. It makes the call soon after establishing the environment. All other functions are used by other managers in the kernel.

The basic functions in the process manager are as follows. **Crtpcb** creates a new process control block. **Set_pcblsts** creates two lists of pcb's; one is called `active_pcb` list (the corresponding processes are running or they will be created), and the other is called `zombie pcb` list (pcb's of the processes that have terminated, but whose effects are not yet cleaned up). Each list is a doubly linked list with a dummy header. **Inspcb** inserts a pcb into a list of pcb's at the appropriate position. The list is sorted on the manager definition pointer value. **Delpcb** deletes a pcb from a list. **Findpcb** finds the pcb, corresponding to a manger definition and cooperation class, in the `active_pcb` list. **Getpcb** gets the pcb, of the process whose **unixid** is given, in the **active_pcb** list. **Getcshpcb** gets the pcb, of the process whose xterm **unixid** is given, in the `active pcb` list.

Getusr creates the login user processes. **Getsvr** is called by the port manager to get the pcb of a server whose manager definition and cooperation class are given. Depending on the creation protocol of the manager, either a new pcb, or an old pcb is returned. **Instsvr** is called when a server has to be instantiated. The pcb for the server must have already been created. **Crtprs** is called to create the process. It gets a free mailbox and sends the mailbox id as a parameter to the new process. The Ultrix system call, *vfork*, is used for creating a child process to run the manager.

Killchld is called at the time of bringing the system down. It kills all active processes and calls **chldterm** to cleanup the side effects of termination for each of them. **Chldterm** is called whenever a process terminates or is killed. It frees the mailbox, and if it is a login process, a new login process is immediately created. The pcb is deleted from the `active pcb` list and added to the `zombie_pcb` list for cleaning up. Then **freepcb** is called. **Freepcb** calls **rslvpts** repetitively until **rslvpts** cannot reslove any more ports. If all ports are cleaned up, the pcb is deleted from the `zombie_pcb` list, the pcb is freed and the process is forgotten for all purposes. Otherwise, **freepcb** is visited on part of the terminated process again from **rvkostdcaps** (see port manager).

Rslvpts attempts to clean up all the ports attached to a process that are in a state to be de destroyed. A port is resolved when all pending requests and outstanding capabilities are revoked and returned to the client and server of the port. If a port is resolved and destroyed, it might

make other ports resolvable. For this reason **rslvpts** returns immediately with **true** in order to be re-invoked by **freepcb**. This repetitions stops when no more ports can be resolved. Then **rslvpts** calls **free owncaps** which releases all the transient non-port capabilities owned by the process.

Noteid simply notes the unix id of a process.

5.4.4 Port Manager

Port Manager module supplies the functions to support all port-primitives. Some of the basic functions are described first, followed by the primitive-functions.

Basic Functions

Newccb creates a new channel control block and initalises some of the fields. **Free ccb** frees the storage occupied by a ccb. **Newccb** is called by **newport** every time there is a request to create a new port. **Newport** creates the user-end and the server-end capabilities for the new port by calling **crtusrptcp** and **crtsvrptcp** respectively. The user-end capability gets added to the transient list of the user. And the server-end capability gets added to the list of unaccepted ports list of the server. **Rmv_uptcap** removes the user-end capability of a port from the user's port list and frees the capability itself. **Rmv sptcap** removes the server-end capability of a port from the server's ports list (unaccepted or transient) and frees the capability itself. It calls **undlvrpcap** to delete a port capability from the unaccepted ports list of the server. It also releases the cooperation class capability associated with the port.

As explained in the section on Data Structures, there are different queues attached to each port. The following functions do various operations on these queues. **Newobj** creates a new object, **enqobj** queues an object, **deqobj** dequeues the object at the front, and **lastobj** dequeues the object at the end of a queue. **Dispose_objq** disposes all objects in the object queue. It dequeues each object and frees all the dangling capabilities (see Data Structures section) in the object. And it also informs the process that sent the object. This is called at the time of destroying the port. **Dispose_dbufq** disposes all the objects in the dbuf queue.

The port capabilities, like any other, can be lent to other processes. They, again like others, have to be returned to their original owners. For this purpose, the kernel maintains a list of all the channels a port capability has crossed over. This list is maintained as a stack. Whenever a port capability is lent over a channel, the channel id is pushed onto the stack and whenever the capability is returned the channel id is popped from the stack. Since either end can be lent, there are two such stacks, called **usrstk** and **svrstk**. These fields are part of the channel control block associated with the port. Whenever port capabilities are being lent over a channel, a **ccblog** node is created for each such capability, with the channel id stored in. All those nodes are put in a list. When the capabilities are actually delivered, each of the ccblog nodes is taken from the list (**popccblog** and pushed onto the **usrstk** or **svrstk** of the channel the port capability belongs to. When the capabilities are actually returned (by invoking Send by the server of the SendReceive port), the **ccblogs** are popped (**popccblog**) from the user or the server stack.

When capabilities are sent (lent) a check has to be made whether sending (lending) process has the capabilities in its possession. **Hascaps** takes in a list of capabilities and checks if each one of them exists in the process's domain. There are some capabilities, called ***exclusive***, that cannot be copied. When an exclusive capability is to be sent, it is kept in a temporary array called exclusive capabilities array. Every time an exclusive capability is to be sent, it is also checked if it is already there in the array (indicating an error). **Hascaps** makes an extra check for the port capabilities. There is an associated cooperation class capability created at the time of creation of a port. If a port capability is being sent (lent), this cooperation class capability is sent (lent) along. **Hascaps** checks if the process has this cooperation class capability. One should note that this capability does not explicitly appear in the message parameters of either the sender or the receiver, but the kernel automatically does the implicit transfer.

Vld_sndcaps and **vld_lntcaps** call **hascaps** to check if the user process has the capabilities it wants to send or lend. Then they call **cons dnglclst** to construct a list of all such capabilities. Each node in this list is of **ostdcap** structure. Unlike the other two, **vld_rcvcaps** checks if there are already capabilities with the same specifications as the process now wants to receive. If it does, then it is in error and the primitive is aborted.

Cons dnglclst constructs a dangling capability list of all the capabilities that are being sent or lent as part of a message on a port. Each node in the list has **ostdcap structure** (see Data Structures section). For Send operation, the dangling capabilities are freed after delivery. But for SendReceive operation, the list is retained as the outstanding list, after the delivery, because they have to be returned to the sender by the receiver at a later time. **Free dnglclst** restores the capabilities (that are not copies), to the original list of the process from whom they were taken.

Dlvr sndcaps takes the dangling c-list, and transfers them to the receiver under the receiver supplied names, and frees the dangling capability list. **Dlvr_lntcaps** also does the same, but without freeing the dangling list, it is made the outstanding capability list. When port capabilities are being lent, the channel id over which they are being lent is registered in the **ccblog** that is pushed onto the user or the server stack of the port. **Dlvrdata** simply copies the non-privilege data into the receiver supplied buffer.

Hard dstrpt destroys a port for good, in the sense that no trace of it appears in the system. It removes the user-end and the server-end capabilities, disposes the object/buffer/details queues and frees the channel control block. **Soft dstrpt** checks if the port is completely clear (no operation in progress, and both the ends are with the original user and server) and calls **hard_dstrpt**. Otherwise, it sets the `dstr onclr` flag.

Getccb is called by almost all the port primitives. Given a port name, this function returns the corresponding channel control block. Depending on the primitive at work, a search is made in different port lists of the process to find the port capability. If it exists, the ccb id of the port is returned.

Rstoreilst is called, when a port operation is completed or aborted, to delete the capabilities, that are part of the port operation, from the intention list of the process.

Rvkostdcaps is called at the time of resolving ports (to clean up the side effects of terminated processes) or at the time of completing SendReceive call. It returns to the owner the capabilities that were lent. **Ostdready** checks if the outstanding capabilities are ready to be returned. To be ready, all the capabilities lent over a channel must exist in the borrow lists of the server and additionally for the port capabilities, all the ports must be clean. This function is called when the server makes the Send to complete SendReceive request. At that time all the capabilities that the server received must be in a position to be returned to the client. This is also called at the time of resolving ports after termination of affected processes.

While requesting various port primitives, the caller may choose to block until a particular event occurs. **Blockon** is called to store the request parameters, request id, the event to block on, and the channel on which the event should occur.

When a server makes a call to accept a port to serve some operation associated with a cooperation class, **matchrqst** is called by a top-level function to find if there is a matching port in the unaccepted port list of the sever.

Notify is called by most of the functions to notify the other end of a port about an event that is triggered by the current process. It checks if the process at the other end is blocked on the current event and if so that process's request is restarted.

Port Primitives

When a process wants to create a port, **gp crtpt** is invoked. The process cannot have another port with the same name as the one given for the new port. It should have the operation capability for the specified operation and the operation should be supported by the manager. **Newport** is called to create the new port, and notify is called to notify the server process (if already created and is blocked on a new port) of the creation. **Gp dstrpt** is called when a client process wants to destroy one of the ports. **Soft_dstrpt** is called and the server is notified of the destruction.

From the server side, when a server wants to accept servicing a new port, **gp acptrqst** is called. It makes sure that the server does not already have a port capability with the same name as the new one. If there is a matching port, its server capability is taken from the unaccepted port list and added to the c-list of the server. The server can decide to reject servicing a port any time after accepting it (but not in the middle of servicing a request) by calling **Gp reject** which calls soft_dstrpt to destroy the port.

Gp revk is called by the client of a port to revoke the most recent request on the port. The queued object is removed, the capabilities are returned to their original places, and the intention lists are deleted. Of course, if the message has already been delivered to the server then it becomes irrevokable. **Gp_rfus** is called by the server of a port to refuse the new request on the port.

The server has two primitives to look at its ports without actually servicing the requests. **Gp_exam** is called to find out the number of messages or the contents of messages on a particular port. The count or the contents are packed into a buffer and sent to the server. **Gp qrypts** is called to query the ports that are under the service of a process. If there are any ports with requests from the clients, the port names are packed and sent to the server. The server may choose to block

until there is a request on at least one of its ports. When the client makes the actual request, notify is called to unblock the server.

Here is a very brief description of the rest of the port-primitives. They are the most commonly used primitives.

Gp send is called by the clients of Send ports, and the servers of Receive and SendReceive ports. It checks if the caller has the port capability and the call parameters are valid. Then the data and capabilities are delivered to the process at the other end. Sometimes, the message has to be queued if the receiving process (for Send and Receive ports) has not yet invoked the corresponding Receive operation. For SendReceive ports, it is made sure that all outstanding capabilities are returned at this time.

Gp recv is called by the clients of Receive ports, and the servers of Send ports. If the sender has already posted the message it is delivered to the caller, otherwise the buffers are queued until the sender responds. **Gp sdrv** is called by the clients of SendReceive ports. The servers of those ports invoke **get_dtls** to get the details of the client request. And later the server completes the call by invoking **gp send**.

5.4.5 Run-Time Support (rts)

So far, only the kernel components have been described. The outside world interacts with the kernel through run-time support (**rts**). This section gives a brief overview of the run-time support and how a process interacts with the kernel.

The run-time support package contains all the functions that are required by a process to set up the environment and invoke the Gutenberg primitives. When the Gutenberg kernel creates a new process, it sends a mailbox id to the process as an argument. Before invoking any Gutenberg primitives, the user process must have setup the environment. **Setenv** is called to open the mailbox, bind the signal handlers, and establish communication with the kernel. The **rts** package contains one function corresponding to each Gutenberg primitive. The user should call a corresponding function in the **rts** package to invoke a Gutenberg primitive. (For the invocation details, see the **rts** section in the User Manual). And before terminating, the process calls **clnenv** which calls **closmbx** to detach the mailbox from the process.

Each of the primitive-serving functions, called stubs, accepts a set of parameters required to perform the primitive. Then the function calls **newmsg** which creates a request block with a unique request id. The request block is filled with the call parameters and **postmsg** is called to send the request to the kernel. **Postmsg** calls **wrtkmbx** to copy the call parameters into the mailbox, logs the request and sends a SIGML signal to the kernel. **Postmsg** then waits for the response (**waitmsg**).

Waitmsg and **readmsg** have a few lines of cryptic code. They use non-local go to statements (**setjmp** and **longjmp**). This is necessary to handle response to asynchronous requests, while the process is currently blocked on another request. **Waitmsg** calls **setjmp** to save the environment to which **longjmp** returns at a later time, and it then pauses. When the kernel sends a SIGML,

the process's SIGML handler, **readmsg** starts executing. **Readmsg** calls **rdsmbx** to read the response into the request block. If the response is for the currently blocked request then it makes a **longjmp** to the previous environment in the **waitmsg** and returns from there to the **postmsg**. Otherwise the control returns to the pause in **waitmsg**. For further clarification on **setjmp** and **setjmp**, one is recommended to refer to the Ultrix Programmer's manual.

5.4.6 Input/Output

A number of I/O functions are written to enhance the Ultrix provided standard Input/Output library package. These functions are made available to the user processes as well. **Getaline**, **getlname**, **getint** and **getch** read a line, an `lname` (local name of a capability), an integer, or a character from the specified input file. **Rd_stdout** redirects the standard output of the process to a specified file.

5.5 Life-Cycle of a Process

In this section, the kernel's workings are described by illustrating the different stages in the life-cycle of a process in the Gutenberg system. The state diagram of a Gutenberg process appears in Figure 5.6.

When an existing process creates a port, a pcb is created for the server that serves the port (assuming that the server does not already exist). The status of the server process is marked as `uninstantiated`. When the client process invokes an operation on the port for the first time, the server process is instantiated from the manager definition, and it enters the `created` state. It invokes the kernel implementation primitive, **noteid**, and enters `active` state. Until a process enters `active` state, it is not under direct control of the kernel. From then on it accepts and/or creates new ports and starts servicing the attached ports. While invoking a port operation, if the process specifies to be blocked until an event occurs, it is kept in the `blocked` state. When such an event occurs, the kernel unblocks the process by calling **notify**.

At the time of aborting the Gutenberg system, the kernel kills the process, and the process enters `zombie` state. Or, in order to bring the system down gracefully, the `active` and the `blocked` processes are put in `quitting` state. They stay in that state, until all processes in `created` state become `active`. The kernel then puts them in `quitting` state. At that point the kernel kills them all.

In any case, a process finally reaches `zombie` state. When all attached ports are cleaned up, the pcb of the process gets deleted from the system and the process becomes `non-existent`.

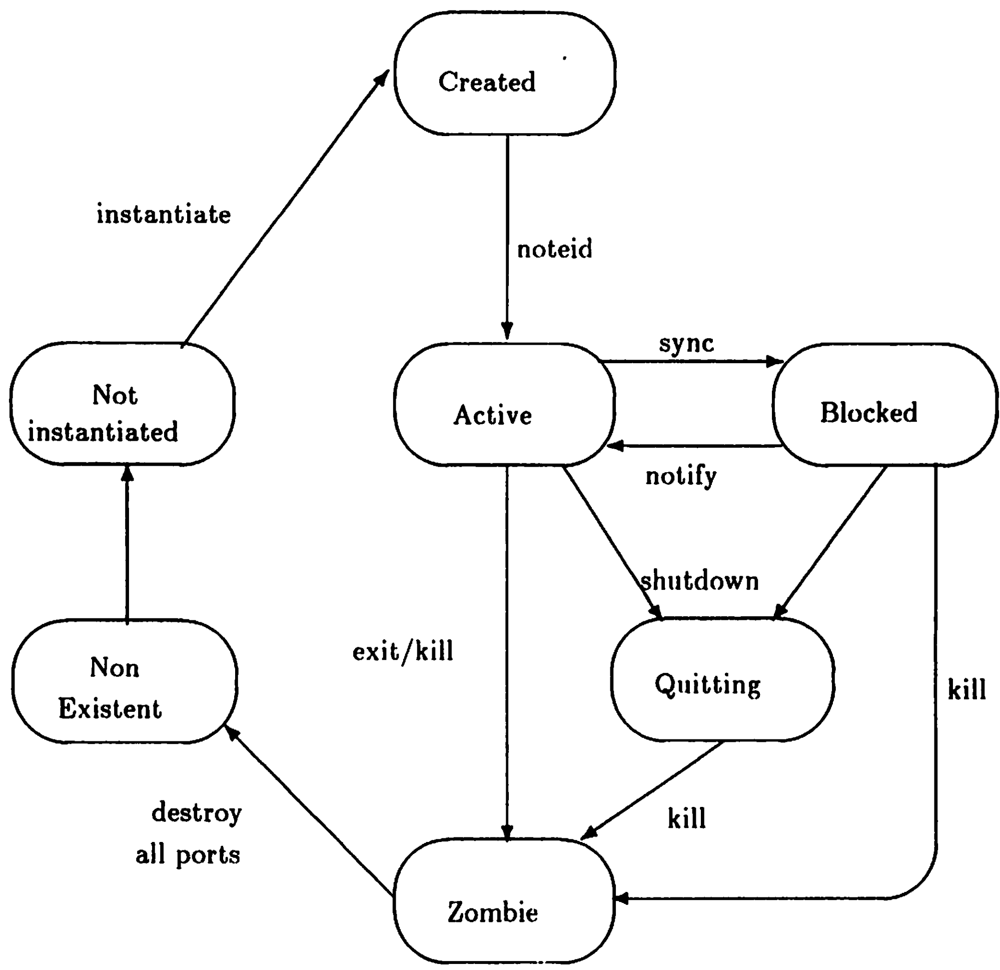


Figure 5.6: State Diagram of a Gutenberg Process

6 System Administration

6.1 Introduction

This chapter is intended to provide the background information about the system initialization and its execution environment.

In the next section, the unix directory structure of the system is described. In the third section, the basic Interconnection Schema structure is presented. The following section describes the *booting* process and the system configuration parameters. The fifth section deals with the building of the Interconnection Schema from scratch. Finally, the kernel monitor/console facility is presented.

6.2 Gutenberg System Directory Tree Structure

The Gutenberg kernel expects a minimal directory tree structure for execution (figure 6.1(a)). This minimal directory structure together with the development directory tree structure constitute the base Gutenberg system directory tree structure (figure 6.1(b)). The base directory tree structure is assumed by the make facility which compiles and links the kernel, the gcp and the servers.

The following paragraphs briefly describe the contents of each directory. A number of auxiliary directories that are used during the development and debugging processes are also described.

6.2.1 Minimal Directory Tree Structure

um2.0/adm is for the use of the kernel. During initialiazation, the kernel looks for the system configuration files and the Interconnection Scheme checkpoint files in this directory.

um2.0/bin contains executable binaries of the system and the user-level managers. These include the kernel executable code (*knl2.0*) and the gcp (*gshl*).

um2.0/exc contains the log (history) files created by the gcp and temporary files created by the system and other applications. This is the default execution directory of the whole system.

um2.0/scr contains the system and user-level script files. The gcp to execute a script, it searches for the file in this directory.

6.2.2 Development Directory Tree Structure

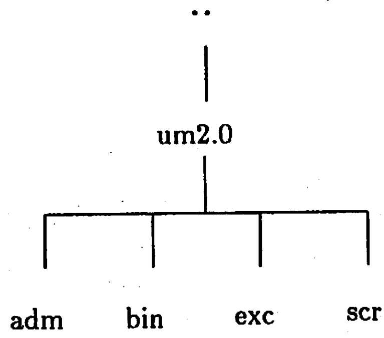
um2.0/cdbc contains the source and object code of the booting and checkpointing module of the kernel.

um2.0/cdm contains the source and object code of the Interconnection Schema (or Capability Directory) manager.

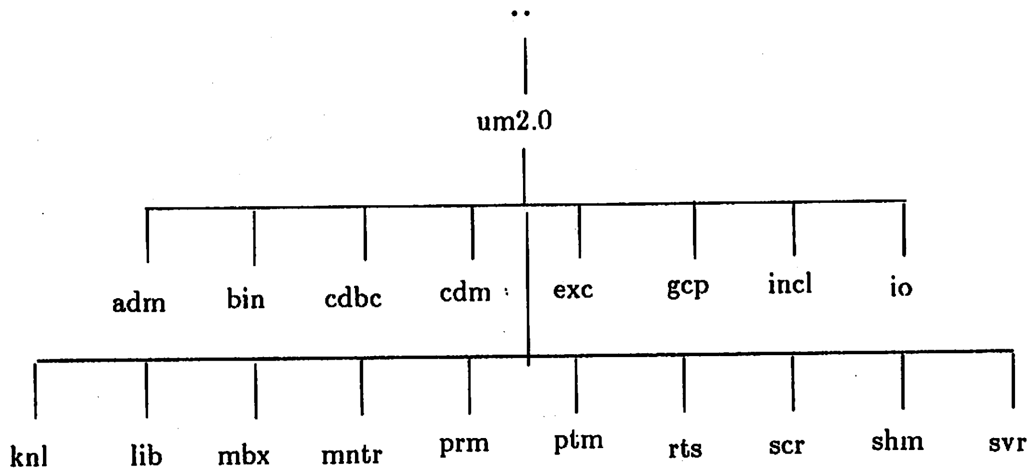
um2.0/gcp contains the source and object code of the gutenberg command processor.

um2.0/incl contains the system include files.

um2.0/io contains the source and object code of the gutenberg specific i/o functions.



(a) Minimal Directory Tree Structure



(b) Base Directory Tree Structure

Figure 6.1: Directory Tree Structure

um2.0/knl contains the source and object code of the kernel control manager.

um2.0/lib contains the system libraries.

um2.0/mbx contains the source and object code of the mailbox facility.

um2.0/mntr contains the source and object code of the monitor facility.

um2.0/prm contains the source and object code of the process manager.

um2.0/ptm contains the source and object code of the port manager.

um2.0/rts contains the source and object code of the run-time-support.

um2.0/shm contains the source and object code of the interface to the unix share memory facility.

um2.0/svr is the root directory of the directories holding the source and object code of the various system and user-level managers. This and *um2.0/bin* are the only directories that applications are allowed to create files.

6.2.3 Auxiliary Directories

um2.0/lint contains the makefile that invokes the unix *lint* command to perform type checking of the kernel and the gcp. The output files are also stored in this directory.

um2.0/ptst contains the source and object code of a test module. It is used for debugging purposes. Gcp alone does not allow interactive port operations. This module, when combined with gcp, allows port operations from the gcp level. The ptst.scr script file sets up the manager definitions for the port server.

um2.0/ptsvr contains the source and object code of a port server to be used in conjunction with the above test module. It can be linked with gcp and one can interactively execute port operations and see their effects. The above two modules are are quite useful for testing the kernel, when one makes changes to the port manager. These modules should be made upto date to reflect any changes done to the rts and gcp.

um2.0/util contains a number of utilities, for example cross-reference utility, for the development and maintainance of the Gutenberg system.

6.3 Interconnection Schema Structure

The Interconnection Schema that the Gutenberg kernel maintains, is similar to that of the Unix file system. It has a root directory within which two other directories are registered, namely, **usr** and **sys** (figure 6.2). That is, the Interconnection Schema has two partitions. One for the users and one for the system.

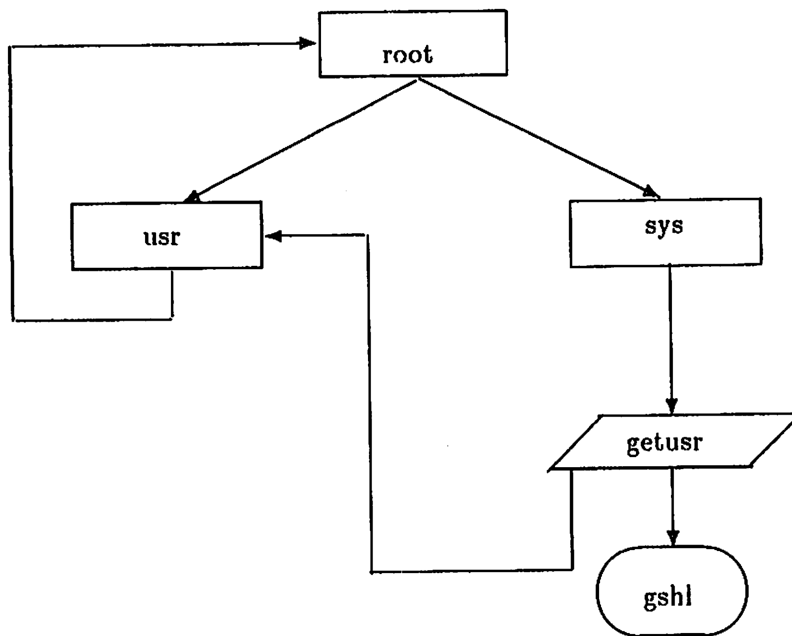


Figure 6.2: Minimal Interconnection Schema

The **usr** directory contains all the users login or primary directories. The **root** directory is also registered in this directory to facilitate the login as a super-user. The super-user's login name is *root*. A new user is added to the system by creating a directory whose local name is same as the username and registering it in the **usr** directory. Only a super-user has the privilege to add new users and navigate the **sys** partition.

The **sys** directory contains the systems directories and system defined managers as, for example, the manager *getusr* which is the manager definition for the login/gcp server.

The Interconnection Schema is expanded by allowing users to create and register new directories and managers.

When the kernel is installed and booted for the first time, it comes up with the minimal structure as shown in figure 6.2 and allows the root user to login.

6.4 Booting Process

In the *um2.0/bin* directory, there is a command file called *gknl* that brings up the kernel. *Gknl*, when executed creates a Xwindow for the kernel's monitor/console and runs the kernel. There are two different modes that the kernel can be set up:
single user mode

```
gknl s
```

and *multi user* mode

```
gshl
```

In the single user mode, the **root** is created along with the **usr** and **sys** partitions in the minimal configuration (figure 6.2). Generally, this mode is used to create the Interconnection Schema the first time or after a crash which has destroyed the checkpoint files. In the other mode, the Interconnection Schema is reconstructed from the checkpointed versions.

During the initialization, the kernel reads the configuration values from the administrator database files stored in the *um2.0/adm* directory. The *sysprm.db* file contains the id of the site, the number of interactive managers (xterms) to be supported (the maximum number is 11, imposed limitation by the unix system), the number of login windows, and the intervals (in sec and ms) between the kernel pollings for lost interrupts. Each of these values appears on a separate line in the order stated above.

The *stty.db* file contains the Xwindows' parameters. Each window is associated with three parameters: its coordinates, the color of its background and the color of its foregraound. Each parameter should appear on a separate line in the order stated above. The number of Xwindow specs should be in agreement with the number of the supported interactive managers in the *sysprm.db*. For details about the format of the coordinates and the available colors, the reader should consult the *Xwindow Manual*.

The last two steps of the initialization are the instatiation of the *getusr* managers that allow a user to login in an interactive session, and the activation of the polling timer or alarm.

6.5 Interconnection Schema Construction

When the kernel is installed and booted for the first time, or is booted after a crash which has destroyed the checkpoint files, the Interconnection Schema needs to be (re)constructed.

In order to construct the Interconnection Schema, the kernel should be set up in a single user mode which comes up with the minimal Interconnection Schema that allows the root user to login. Then the super-user can construct the Interconnection Schema either manually by login and typing each and every command, or semi-automatically by using the script command of the gcp. For example, the current version maintains 6 such script files in the *um2.0/scr* directory:

addusers.scr creates a login directory for each user of the system.

usrpart.scr (re)constructs the users directory structures.

filesys.scr installs the gutenberg file system manager.

qrydb.scr installs the database query application

qryfls.scr installs the database query environment and the database files.

ggame.scr installs the guess game.

The *usrpart.scr* can be viewed as the system backup file. It depends on the users' directories and that is why it should be executed after the *addusers.scr*. For similar reasons the *filesys.scr* should be executed before the other scripts and the *qrydb.scr* before the *qryfls.scr*. The order of *ggame.scr* and the query scripts can be anything. That is, the order of execution of the scripts is defined by the dependencies between them.

The scripts can be created either from the log file(s) after constructing the schema manually, or by using an editor.

6.6 Monitor Facility

The monitor facility is provided to the system administrator and developers for administrative purposes and to examine the internals of the Interconnection Schema.

The system administrator can invoke the monitor by pressing *ctrl-c* at the kernel console (process window).

The system administrator can use the **monitor** to traverse through the Interconnection Schema, take checkpoints, or bring the system down. The **monitor** provides different ways of bringing the Gutenberg system down. *Abort* option is used to abruptly exit the system. With this option, all managers are immediately terminated, and the kernel exits without taking any checkpoints. *Exit* option terminates all the managers, brings the Interconnection Schema to a consistent state, and takes a checkpoint optionally. *Shutdown* option is used to give a warm start to the kernel.

Here is a list of the monitor options:

a: aborts the kernel and terminates all managers.

d: displays the current directory. Initially **root** is the current directory.

e: exits the kernel after terminating all managers and optionally takes a ckeckpoint of the Interconnection Schema.

g: instantiates a *getusr* manager to allow one more user to login. This bypasses the restriction on the number of interactive users defined by the system configuration parameters.

m: displays a manager.

q: resumes kernel's normal execution.

s: (shutdown) warm starts the kernel. The interconnection schema is not affected.

t: traverses the Interconnection Schema. It prompts for a path that can be absolute starting from the **root** directory or relative staring from the current directory. A path is a concatenation of directory names separated by '/'. A directory name is the concatenation of the names of the directory and its cooperation class separated by '|'.

w: changes the window configuration. It reads the new Xwindows' parameters from a specified file in the same manner as in booting. That is, the specified file must have the same format as the *stty.db* and exist in the *um2.0/adm* directory.

APPENDIX A

Gutenberg Error Manager

V. R. Govindarajan
Panayiotis K. Chrysanthis

1 Introduction

This document describes the salient features of an error server implemented in the Gutenberg environment. In Gutenberg, whenever invocation of a primitive results in an error, the user is returned an error number. This server performs the function of returning an error message corresponding to the error number supplied by the user. The server is implemented as a Gutenberg manager, and the primitives provided by Gutenberg are used in implementing the server. The reasons for implementing the server outside Gutenberg Command processor (gcp) are the following:

1. The server need not always be a part of the gcp. Only when required, does the server need to be invoked. This keeps the size of gcp small.

2. If the server is part of gcp, it cannot be invoked by any other application.

3. This was the first server implemented on top of Gutenberg, and it illustrates the ease with which servers can be built in the Gutenberg environment, and also tests the functionality of primitives provided by Gutenberg.

2 Algorithm

The algorithm employed to find an error message corresponding to an error number is based on index sequential searching of a file, where indices are not permanently stored. A first pass through the file containing error numbers and messages is made to build the indices. The indices in this application are offsets of error numbers from the beginning of file. The offsets of error numbers which are multiples of a pre-defined number (MULTIPLE) are stored in a table *error_index* so that faster access to the error file is made possible.

When a user supplies an error number, the file pointer into the error file is positioned to the error record closest to the given error number for which an offset is available. Then the error records are sequentially scanned until the requested error record is obtained. The error message is then returned.

3 Data Structures

The error numbers and corresponding messages are stored in the file *gtnerr.h*[1]. The error number is a negative three digit number. In the present implementation, an error message is assumed to

[1] In the present implementation, all files are directly accessed via Ultrix

be of maximum size 60 characters, and is defined by the constant ERRMSG_SZ. (This is not a rigid assumption, and it can be easily changed, if necessary.) The offsets of all error numbers which are a multiple of MULTIPLE are stored in the table *error index*. (In the present implementation, MULTIPLE is defined to be 20.) The error message corresponding to a given error number is returned in *error buf*.

4 Specifications

Type: A conservative manager which does not depend on its attached ports.
Exports: *askerr*, an operation returning an error message corresponding to a supplied error number is invoked by means of an SendReceive primitive. The error number contained in request details is an integer that can have a maximum value of 999 with an optional leading minus sign. After the error number is supplied, theappropriate error message is returned to the user.
The syntax of the operation is:

```
SendReceive (port, errno, INTSIZE, errmsg, ERRMSG_SZ,sync, status)
CPSPECS *port ; /* port used for invoking this operation */
int *errno ; /* error number for which message is required */
char *errmsg ; /* buffer in which error message is returned */
int sync ;
int *status ;
```

The above syntax is for invocation of the server with in a manager. If the Error Manager is to be invoked interactively from gcp, the invocation is *operation local name, coop_class*, as defined by *external* command, where *coop class* capability can be of any name.

5 Pseudo-Code

In this section, the pseudo code of all routines used for implementing the server is presented. Gutenberg port primitives used for realizing the mentioned operations are given wherever applicable.

1. **main program**:
   /* Initializes environment and invokes other procedures to serve the user */

   ```
   Initialize environment ;
   repeat
       Perform error server function ;
   forever ;
   ```

2. **initialise**:
   /* Make first pass through *gtnerr.h* and get indices of error numbers */
   open *gtnerr.h*
   while (not eof(*gtnerr.h*))

read error number ;
if (error number is a multiple of MULTIPLE)

Store index into ***error_index*** ;

close ***gtnerr.h***

3. **errfnd:**
/* Gets the error number and returns corresponding error message */

Set up a port for getting error number, and returning error message ;
/* ***acptrqst(ptname, operation, group, ...)*** */

Get error number over the port in a buffer ;
/* ***getdetls(ptname, dtl_bufptr...)*** */

Convert details got into an error number ;
Invoke ***getmsg()*** ;
/* to get error message corresponding to this number */

Send error message over the port to the user ;
/* ***send(ptname, ...)*** */

Destroy the port ;
/* ***reject(ptname...)*** */

4. **getmsg:**
/* get n, the error number, and return corresponding error message */
Open ***gtnerr.h***
Convert error number into a positive integer ;
Position file pointer to an error record for which an index is available and which is closest to the supplied error number ;
Sequentially scan ***gtnerr.h*** until the required error number is obtained, or it is found that the required error number is absent in the file.
Return appropriate message.

APPENDIX B

```
/***************************************************************************
 *                                                                         *
 *  Gutenberg Prototype Operating System  -- Umass/Amherst                 *
 *                                                                         *
 *  NAME:        Gutenberg Error Manager                                   *
 *                                                                         *
 *  DATE:        15 April 1987                                             *
 *                                                                         *
 *  AUTHORS:     V. R. Govindarajan                                        *
 *                                                                         *
 *  MODULE:      Utility Managers                                          *
 *                                                                         *
 *  DESCRIPTION:                                                           *
 *               This manager accepts an error number and returns the      *
 *               corresponding error message.                              *
 *                                                                         *
 *  MODIFICATIONS:                                                         *
 *                                                                         *
 *                                                                         *
 ***************************************************************************/

/*
 *  include files
 *
 */

#include "shell.h"

/*
 *  constant definitions
 *
 */

#define MULTIPLE  20          /* error index in multiples of 20 are stored */
#define MAXINDEX  1000/MULTIPLE   /* This is the size of array error index */
#define MAXCHAR   81             /* max number of chars in an error record */
#define MSG_START 32             /* error message starts from this positon */
#define ERR_START 22                       /* error number starts from here */
#define GTNERR    "../incl/gtnerr.h"     /* file storing the error messages */
```

```
#define ERRMSG_SZ 60            /* max number of chars in the error message */
#define START     0             /* to start lseeking from the beginning */


/*
 *  global variables
 *
 */

int error_index[MAXINDEX] ;                /* error number record kept here */
CPSPECS ptcapid, grpcapid ;           /* specification of the request port */
lname operation ;                              /* name of the operation */

/*
 *  main program
 *
 */


main(argc, argv)
int argc ;
char **argv ;

{
   setenv(argc,argv) ;

   initialise() ;                  /* initialise variabless and enters */
                                   /* record number into  error buffer */

   while (TRUE)         /* it is a conservative manager and loops forever */
    {
       errfnd() ;       /* accepts error number and returns error message */
    }

 }


 /*
  *   Error server initialization.
  *
```

```
 *   This function enters the record number of errors which are multiples of
 *   20. This function also initialises all the shared variables.
 *   This is called by main.
 *
 */


initialise()

{

   FILE  *fp ;
   char onerecord[MAXCHAR], s[3] ;
   int j=0, i=0, n=0, len = 0, prevn = 0 ;

   for (i=1; i<MAXINDEX; i++)
     error_index[i] = -1 ;

   error_index[0] = 0 ;                           /* error 0 is first record */
   i = 1 ;

   fp = fopen(GTNERR, "r") ;
   while (fgets(onerecord, MAXCHAR, fp) != NULL)
    {
      for (j=0; j<3; j++)           /* copy error num from onerecord into s */
      s[j] = onerecord[j+ERR_START] ;

      n = atoi(s) ;                               /* get equivalent integer */

      if ( (n % MULTIPLE == 0) && ( n > 0) && (n != prevn ))
        { error_index[i] = len  ;
          i++;
        }
      prevn=n;
      len+=strlen(onerecord);        /* has the number of bytes read so far */

     }

    fclose(fp) ;
```

```
  /* copy portname and group id */
  strncpy(ptcapid.name, "err_port", LNSIZE) ;
  strncpy(operation, WILDCARD, LNSIZE) ;

  return ;

}

/*
 *  Error server
 *
 *  This function accepts a request, gets the error number sent as part of
 *  the request details and returns the corresponding error message. It
 *  accepts a port for a single request so it rejects the port after
 *  satisfying the request. In case that, the port is not immediately
 *  destroyed but it is marked destroy-on-clear, a new default port name
 *  is generated to be used for naming new ports.
 *
 */


errfnd()


{
   int status;
   int dtl_size = 10 ;                          /* size of detail buffer */
   int i = 0, j = 0 ;                                /* local counters */
   static int counter = 0 ;                 /* for creating new port name */
   long n = 0 ;

   char dtl_buf[10] ;                          /* details received here */
   char error_buf[ERRMSG_SZ] ;                 /* error message buffer  */
   char *err_bufptr = error_buf ;             /* pointer to error buffer */
   char *dtl_bufptr = dtl_buf ;             /* pointer to details buffer */
   char charnum[10] ;
```

```
        for (j=0; j<ERRMSG_SZ ; j++)
          error_buf[j] = ' ' ;

        strncpy(grpcapid.name, WILDCARD, LNSIZE) ;

        /* accept request -- synchronous */
        acptrqst(&ptcapid, &grpcapid, operation, TRUE, &status) ;
        if (status != SUCCESS)
          {
           printf(" ERROR: return status%d\n", status) ;
           return ;
          }

        /* get details in dtl_buf */
        getdtls(&ptcapid, dtl_bufptr,&dtl_size ,TRUE, &status) ;
        if (status != SUCCESS)
          {
           printf("ERROR: return status%d\n", status) ;
           return ;
          }

         n = atoi(dtl_buf) ;

        /* get error message */
        getmsg(error_buf, ERRMSG_SZ, n) ;

        /* send error message to the port */
        send(&ptcapid, err_bufptr, ERRMSG_SZ, FALSE, TRUE, &status) ;

        /* reject port */
        reject(&ptcapid, &status) ;
        if (status != SUCCESS)          /* get a new port name */
          {
           gcvt((double)counter++, 8, charnum) ;
           strcpy(ptcapid.name, charnum) ;
           return ;
          }

    }
```

```
/*
 *  get error message
 *
 *  gets the error number from errfnd and returns the  error message
 *  in err_buf found by indexing in the file storing the error messages.
 *
 */

getmsg (err_buf, err_size, n)
int err_size ;
char err_buf[]  ;
long n ;

{
  static int fd = -1 ;
  int j = 0, i = 0, err_num, len = 0 ;
  char onerecord[MAXCHAR] ;
  char *rbuf = onerecord ;
  int num = 1 ;
  char s[3] ;


  if (error_index[n/MULTIPLE] == -1)            /* error number not defined */
    {
     strncpy(err_buf, "Error number not defined", 26) ;
     return ;
    }

  /* open only once, for the first time */
  if (fd == -1)
    fd = open (GTNERR, 0) ;

  if ( n < 0 )
    n = -n ;                                    /* convert neg n to positive */

  lseek(fd, error_index[n/MULTIPLE], START) ;
                                      /* position fd to the requested record */
```

```
    err_num = -1 ;

    /* the following code checks for the validity of the given error number */

    while (err_num < n )
     {
      for (i =0; i<MAXCHAR ; i++)
      onerecord[i]=' ';
      rbuf = onerecord ;
      i = 0 ;
      while (onerecord[i] != '\n')
       {
        rbuf++ ; i++ ;
        read(fd, rbuf, num) ;
       }

      for (i = 1 ; i < 4 ; i++)
      s[i-1] = onerecord[i+ERR_START] ;                        /* copy number */

      err_num = atoi(s) ;

      if (err_num > n)
        {
         strncpy(err_buf, "Error number not defined",26) ;
         return ;
        }
     }

    /* copy message into err_buf */
    j = 1 ;
    /* output till end of comment */
    while ((onerecord[j + MSG_START] != '\n') && ( j < err_size))
     {
      if (! (onerecord[j+MSG_START] == '*' || onerecord[j+MSG_START] == '/'))
      err_buf[j-1] = onerecord[j+MSG_START] ;
      j++ ;
     }

}
```

APPENDIX C

```
# This Script creates a manager definition for the Gutenberg Error Manager
#
crt cc      % creates a transient cooperation class capability with the
errsvr      % "errsvr" name to be used for the file containing the manager's
            % object code.
#
crt dir     % creates a directory with name "errdir" to be used as the
errdir      % manager's initial active directory.
#
# crt mgr creates the actual manager definition called "error"
crt mgr
error       % name of the manager.
0           % instantiation protocol Conservative (0).
n           % not interactive; executes in the backround.
errsvr      % name of the object file (cooperation class capability).
errdir      % name of the manager's initial active directory
            % name of the cooperation class that the directory belongs to (NULL).
1           % number of exported capabilities.
askerr      % name of the exported capability.
X           % port type to be used for requesting the capability: SendReceive.
1           % number of parameters in the respond message.
0           % number of capabilities in the respond message.
1           % number of parameters in the details.
0           % number of capabilities in the details.
60          % size of respond parameter in bytes.
10          % size of the details parameter in bytes.
#
# crt op creates an operation capability linked to the Gutenberg Error Manager
crt op
askerr      % name of the operation
error       % name of the manager exporting the operation
            % name of the cooperation class that the manager belongs to (NULL).
askerr      % generic name of the operation (known by the manager)
#
# terminates the script
quit
```