GARD: A UNIFORM INTERFACE TO SYSTEM RESOURCES

Rob Lake
Allan Christie
Gord Urquhart
Dale Hagglund

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

Conventional operating systems have several types of *resources* (e.g. files, devices, processes, memory, etc.) which the user accesses. The set of operations performed on a resource usually differs for each type and, consequently, the user must know the type of resource being used. Ideally, it is desirable to have all these resources handled in a uniform manner.

This paper introduces a capability-based system called **GARD** which attempts to remove this user level distinction of resources. We present a uniform user interface which allows the user to have minimal knowledge of system resources. This is followed by a description of how the kernel handles the wide variety of resources present within the system.

KEYWORDS: capability, object, primitive operation, uniform interface.

1. Introduction

What is the difference between a file and virtual memory? Conventional operating systems emphasize the distinctions between these and other resources, and make little attempt to hide these differences from the user. Conceptually, files and virtual memory may be regarded at the user level as being identical resources, since both are used for storing data. We believe these differences should be minimized at the user level so that users need not be aware of the idiosyncrasies of the various resources. Furthermore, users should be equipped with only one set of operations to handle all resources.

While some attempt has been made by newer systems to provide a uniform interface to resources, most do not carry this concept far enough. Users are still required to have some knowledge about the resource (and consequently, the set of operations) being used. A different interface is needed to allow the user to have minimum knowledge about the resource type, know only about one set of operations, and have the operating system handle the various distinctions between resources.

This paper presents an interface to an operating system which hides from the user the distinction between system resources, and provides a single set of resource operations.

2. Capability-Based Systems

Capability-based systems [2, 5, 6, 10] provide an approach to defining a uniform user interface. These systems provide a single mechanism for addressing both primary and secondary memory, and for accessing both hardware and software resources. Each user is equipped with a set of *capabilities* which govern the degree of access the user has to all objects on the system.

A capability is a key or token which gives the user some form of permission to a particular object. Capabilities are represented as data

structures and consist of two parts: a *unique object identifier*, and a set of *access rights*. The object identifier represents a single logical or physical entity in the system. An entity may be a file, memory segment, array, process, or device. The access rights describe the set of operations which may be performed on the object.

Capability-based systems differ from conventional systems in the method by which objects are referenced. In conventional operating systems, the list of permissions users have to an object is stored with the object. Capability systems, on the other hand, have protection information stored in the capabilities to the object. This implies users may only *name* or reference objects *to which a capability is held*.

One of the problems with a capability-based system is that once access to an object is given, it is very difficult to remove that access without creating a duplicate of the original object, and then destroying the original. This solution is impractical, since it requires knowledge at all times of who has been given permission to the original object. In this paper, we introduce a derivative of pure capability-based systems which removes this problem.

Capability systems support what is known as the *object-based* approach to computing. This approach starts by defining all entities as *objects*. Next, the set of objects is subdivided into various *classes*, with each class composed of a set of objects containing similar properties. Each member of a class is referred to as an *instance* of the class. Each class can have a set of operations defined which may be performed upon the instances.

The object-based approach has several advantages. First, by defining a set of fundamental objects and operations which may be performed on them, a set of procedures can be constructed to perform these operations for each object type. More importantly, this approach serves to raise the abstraction level of the system. All operations and references may now be made on a higher-level class of data types.

The primary reason for deciding to use a capability system in our attempt to achieve a uniform system interface is the extra level of indirection provided by capabilities. Instead of accessing an object directly, the object is accessed through its capability. This allows for all user-level operations to appear uniform, regardless of the object the capability refers to.

3. User Overview of GARD

Key concepts in **GARD** are those of an object and a capability. All system and user resources are objects, which are accessible only through capabilities. Users interact with these resources through a set of common primitives, each of which can be used on any object.

In **GARD**, as in most capability-based systems, operations are performed on objects through *capabilities*. Without a capability to an object (i.e. resource), there is no way to access that object. This implies that the arguments to the primitives described below are actually capabilities to the objects in question. Each resource in the system, whether active or passive, has access to a set of capabilities which define what other resources it may access. If the capability set for a resource does not contain a given capability for another resource, it is impossible for the first resource to reference the second in any manner.

It should be noted that this, if followed logically to its extreme, encompasses many of the current structures common in operating systems. For instance, a hierarchical file system becomes nothing more than a set of resources where those resources corresponding to directories have capabilities for several other resources which may themselves be ‘‘directories.’’

3.1. Primitive Operations on Objects

If one examines the different types of resources available in conventional computer systems, one finds that the operations performed on these resources fall into a few basic categories. For each type of resource, it is necessary to be able to *create*, *delete*, *read data from*, and *write data to* that resource.

These represent four basic operations on objects. Since all operations on objects are performed through capabilities, from a user’s point of view, destroying an object is equivalent to simply invalidating a capability to it. For this reason, *delet*ing an object is regarded as an operation on a capability, and is described in more detail in the following section.

GARD views a read operation as obtaining data from a source object and placing it in a

---

G. Urquhart, **A**. Christie, **R**. Lake, and **D**. Hagglund.
Although we refer to **GARD** as an operating system, we make no attempt in this paper to fully define all parts of such an operating system. What we describe here is the interface of the system to resources.

destination object. A write operation is seen as taking data from a source and putting it into a destination. It is clear from these definitions that read and write operations are essentially identical. These operations may be considered as identical in **GARD** because the system handles the details of the transfer, rather than having the user deal with buffers, etc. Thus, we may replace distinct *read* and *write* operations with a single *copy* operation.

This leaves us with the following primitives: *create* and *copy*.

Create takes as its single parameter the capability id of an object to act as a template. Returned is a capability to a new object which has the same behavior as the template under the primitive operations. This requires some objects exist *a priori* at system start-up time to act as initial templates. The behaviors of these initial templates implicitly define different types of objects.

Copy obtains data from a source object and places it into a destination object. Each *copy* requires capability ids for the source and destination objects. In addition, several other parameters may be specified. These give the starting offset within the source and destination, and the amount of data to be transferred.

In addition to these, **GARD** defines two other primitive operations on objects that extend the versatility of the primitive operations mentioned above. *Duplicate* makes an exact duplicate of an object, including its capability and access information. A capability id to the duplicated object is returned. *Start* places an object on the CPU queue. When the object terminates, it is deleted from the system.

Thus, the primitive operations on objects defined in **GARD** are: *create*, *copy*, *duplicate*, and *start*.

3.2. Primitive Operations on Capabilities

Like objects, capabilities have a certain set of primitive operations defined upon them. These operations define how objects in **GARD** make use of these capabilities. The primitive operations for capabilities are: *delete*, *pass*, and *permit.*

Delete removes the capability identifier from the invoker's capability list. If no other capabilities for the object remain, the object itself is deleted.

Pass takes a capability from one object and copies it into the capability list of another object. *Pass*ing is the means by which capabilities spread through the system.

The *permit* operation on capabilities allows the holder of the capability to modify any of the permissions to the object. The holder of the original capability to an object returned by *create* is always allowed to change the permissions to the object, regardless of the current permissions. A capability owner not having *permit* access to an object can never change his or any other permissions to the object.

3.3. Access Rights

Holding a capability to an object does not necessarily imply unlimited freedom to manipulate either the object or the capability. **GARD** defines a set of *access rights* that control the scope of operations which may be performed upon the objects and capabilities. Such permissions allow more refined control over objects and capabilities.

Access rights fall into two categories: those that control operations which change the contents of an object, and those that control how the capabilities to that object may be manipulated. The presence or absence of an access right determines whether the corresponding operation can be performed through that capability.

Access rights on objects are:

- *create* — the object may be used as a template for *create*.
- *copy from* — the object may be used as the source of a *copy* operation.
- *copy to* — the object may be used as the destination of a *copy* operation.
- *duplicate* — the object may be duplicated.
- *start* — the object may be placed on the CPU queue.

Access rights on capabilities are:

- *pass* — the holder is allowed to pass the capability to another object.
- *permit* — the holder is allowed to change the permissions on the object referred to by the capability.

These access rights allow users to control the spread of capabilities through the system, as well as to protect themselves from malicious or careless users.

4. The GARD Kernel

So far, a user level description of the fundamental operations on objects has been presented. This section examines these primitives from a kernel perspective and describes the internal implementation of the primitive operations for various system resources.

4.1. Internal Organization of System Resources

GARD defines an object to be an abstraction of a system resource. Although objects may represent a wide range of resources, all objects have a similar structure. Every object in the system has a memory image defining it. This memory image is the same for all objects and consists of five parts:

- *capability list pointer*,
- *access list pointer*,
- *bookkeeping information pointer*,
- *switch table*, and
- *data part*.

The capability list pointer points to a list containing all the capabilities the object owns. Each capability id consist of two fields. The first is an object identifier which uniquely identifies any object in the system. The other is a flag denoting if this is the original capability returned when the object was created.

The access list pointer points to a structure which stores the capability identifiers for all objects having access to this object, and their respective permissions.

The bookkeeping information pointer points to a table containing the object size, the object's virtual location in secondary storage, and other information needed to handle the object correctly.

The *switch table* defines the effect of each of the primitive operations on the object. This table allows the system to map each primitive operation into the appropriate task, depending upon the object. Each entry in the table consists of a capability id referring to an object which performs the operation. Objects have the same switch tables as those in the templates used for their creation.

The switch table mechanism provides the basis for abstract objects since each object contains the necessary information to describe its behavior. This mechanism could allow for the use of user defined objects and user defined reactions.

The final portion of an object's memory image is the data part. This is the only part of the object which may be directly modified by a non-system object. All other parts are protected by the system. The data part of the memory image is accessed through a page table like structure. Each entry either points to a page of data in memory or is invalid. When a data reference is made to a page marked invalid, the page containing the data must be brought in from secondary storage. Otherwise, for valid pages, the image in main memory is used. Standard paging algorithms can be used to keep the most heavily used pages in primary memory. Since an object can become arbitrarily large, it may be necessary to have multiple levels of indirection in the page table.

4.2. Capability Address Translation

Since all objects in the system, whether they be in secondary or primary storage, have the same form of unique identifier, there is no need to make a distinction between objects in primary and secondary storage. This means the same mapping scheme may be used for all objects.

GARD uses a *system mapping table* to perform the translation of a capability into an object location. The system mapping table contains information required to access an object if one has a capability to it. Each table entry has:

- information to map a capability to an object's location,
- a share count of the number of capabilities to this object.

The share count is used for disposing of unused objects. Each entry in the mapping table refers to a particular object. Whenever a capability for an object is deleted, the share count is decremented by one.

The system mapping table is also referenced whenever the share counts of an object are changed. An object is destroyed by deleting its entry from the table. Invalid capabilities are identified when the search of the system mapping table for the capability fails.

When an object is accessed through a

In a conventional system "objects" that are in primary storage or in paging storage are identified by an address, whereas objects in secondary storage are identified differently.

capability, the object identifier is hashed to produce an index into the system mapping table. The access list for the referenced object is checked to see if the invoking object has the requested permissions. If so, the operation proceeds normally; otherwise it fails.

This method requires searching the access list on every reference to the object. Obviously, this requires much overhead, especially for lengthy access lists and frequent references. To improve performance, each capability is given additional fields. These are the time of the last access check and bits indicating the specific permission for each operation. Whenever a new capability is created, these fields are initially zero. Similarly, a time stamp field is added to each entry in the system mapping table. This field records the last time any permissions of the object were changed. This could be an actual clock value or a simple count.

When an access to the object is attempted, the capability time stamp is compared to the object time stamp. If the value in the mapping table is greater than the time stamp in the capability, the permissions on the object have changed and must be rechecked. The access list is searched, the access bits in the capability are updated, the time stamp from the mapping table is copied into the capability, and the operation proceeds if the required access is present. When the value in the mapping table is less than or equal to the time stamp in the capability, the permission bits in the capability are checked for the required access.

4.3. Object Classes

Objects are implicitly divided into classes by their behavior. Each object's behavior is imposed upon it by its switch table. Objects that behave similarly are members of the same class. Since an object can only be created based on a template, the number of different object classes is limited to the number of different templates supplied by the system. This makes the set of object classes easily extensible, since all that is required for a new class is the addition of a new template with the required behavior.

4.4. Internals of the Primitive Operations

This section gives a kernel perspective for each of the primitive operations described earlier.

4.4.1. Object Creation

Whenever an object is created, several operations are performed by the system. First, an object is created with the same switch table as the template's, provided the invoker has *create* permission to the template. A unique capability id is then assigned to the newly created object, and an entry is allocated in the system mapping table for this capability. Next, the capability id of the creator is placed in the access list of the new object, indicating that the creator has unlimited access. The share count for this object is initialized to one, and the location of the object is recorded in the system mapping table.

4.4.2. Capability Deletion

Deletion of a capability removes it from the capability list of the invoking object. The share count of the object referred to by the capability is decremented by one. If the share count becomes zero (i.e., there are no more capabilities to the object) the object itself is destroyed. The destruction of an object causes the system to update the system mapping table, thereby removing the entry for that object. Any primary or secondary storage being used by the object is reclaimed and all capabilities owned by the object being destroyed are deleted. In the case of ports, the capabilities referencing any outstanding messages are also deleted.

4.4.3. Object Execution

Object execution is accomplished via the *start* primitive. *Start* requires four parameters. The first is the capability id of the object to be executed, and the remaining three are capability ids of objects to be used for input, output, and errors. The last three parameters are placed in the capability list of the object, and the object is placed on the CPU queue. When execution terminates, the object is destroyed.

4.4.4. Object Duplication

Duplication of objects is done with the *duplicate* primitive. A new object is created and the entire contents, i.e., switch table, capability list, access list, bookkeeping part, and data part, are copied into the new object. The access list of each object on the original object's capability list is changed to allow the new object the same access as the original. *Duplicate* returns a capability to the new object. This capability differs from that which would be returned by *create* in

that it does not grant the holder automatic *permit* access.

4.4.5. Object Input/Output

Input/output is accomplished with the *copy* primitive. As many as five parameters may be specified:

- the capability id of the source object,
- the capability id of the destination object,
- the offset within the source object,
- the offset within the destination object, and
- the amount of data to transfer in bytes.

If no amount is specified, the entire source object is copied. For certain classes of objects, the amount of data to transfer may be ignored.

The switch table entries for the source and destination objects are consulted to determine how to transfer data between the two objects. These routines are passed the starting offsets in the source and destination objects, and the amount of data being moved.

4.4.6. Capability Id Passing

Capability ids are passed from one object to another using the *pass* primitive. The specified capability id (which must be in the capability list of the object calling *pass*) is copied into the capability list of the destination object.

4.4.7. Modifying Object Permissions

Modification of object permissions is done using the *permit* operation on capabilities. *Permit* operations proceed as follows. The access list of the object whose permissions are to change is checked for an entry giving the invoker this right. If the right exists, the access list is searched for an entry corresponding to the object receiving the new permission. If an entry for this object cannot be found, a new entry is formed with the appropriate permissions; otherwise the old permissions are modified.

Note if the object invoking *permit* has the original capability to the object whose access list is to be modified, the operation will succeed regardless of the current permissions.

5. Behavior and Interaction of Object Classes

We now examine in more detail, the behavior and interaction of various classes of objects. Four common classes are considered: data, devices, ports, and executing objects.

Data objects have their data parts referenced as an array of bytes. Any further structure must be imposed by the user. A *copy to* or *copy from* a data object is a direct access in the array of bytes.

The data part of a device object contains the buffers needed by the device. Each device has either an input buffer, an output buffer, or both. The *copy from* and *copy to* operations work on the input and output buffers respectively. These operations correspond to the upper half of a device driver in that they do not perform the physical I/O.

Ports are the primary means of communication in **GARD**. They provide a uni-directional flow of information. Each port's data part is organized as a list of capabilities, each of which identifies a message. The message queue is processed in a first-in, first-out order. Copying to a port involves taking the data given, placing it in a new data object, and appending the data object's capability to the message queue. A *copy to* may suspend the caller if the message queue exceeds a certain length. A *copy from* operation removes the requested amount of data from the port, or suspends the caller if insufficient data is available. *Copy from* is not required to take all data from a particular message. Once the data has been completely extracted from a message, the message's capability is deleted (which results in the destruction of the object holding the message).

Executing objects are those which receive CPU service. Data parts of these objects contain the code of the executing "process". The copy operations on an executing object are identical to those on a data object. Even though the system allows this, it will most likely result in an execution error in the affected "process". This can be prevented by removing write access to the executing object.

Communication between executing objects is done via ports. Consider the case of setting up a pipeline between two executing objects. The

---

In the case of an executing object duplicating itself, the *duplicate* call returns twice — once in the parent and once in the child. The return value in each case is that to the new object, from the point of view of each object in *which the duplicate returns*. This implies that each object receives a capability to the other.

following sequence of primitives may be used:

```
(1)  portid = create(port_template);
(2)  A = duplicate(A's_object_code);
(3)  B = duplicate(B's_object_code);
(4)  start(A, input, portid, errorA);
(5)  start(B, portid, output, errorB);
```

In this example, (1) creates a new object which behaves identically to the template (i.e., a port), but is initially empty. Lines (2) and (3) create duplicates of the code for objects *A* and *B*. Lines (4) and (5) assign the input, output, and error objects to *A* and *B*, and start them executing. Now, when *A* copies data to its output, the data is placed in the port, from where *B* retrieves it when it copies from its input. *A* and *B* are deleted when they finish execution.

Another common operation on executing processes is a *fork*. Forking a single executing object into two may be done by using *duplicate* since this primitive creates an exact copy of the object given as its argument. The sequence of primitives required appears below:

```
(1)  other = duplicate(parent);
(2)  if (other == parent) {
(3)          child_code();       /* in child */
(4)  } else {
(5)          parent_code();      /* in parent */
(6)  }
```

Line (1) effectively accomplishes the fork. Line (2) determines which half of the fork we are. Since *duplicate* has returned the capability of the other side, we can check to see if the other side was the parent.

6. Other Approaches

Several attempts have been made to handle resources in a more uniform manner. MULTICS [1, 8] succeeded in removing the user distinction between files and virtual memory. However, MULTICS was not able to handle all resources uniformly.

UNIX® [7] enables the user to access a common set of system subroutines for performing operations on devices, files, and pipes connecting executing processes. However, resources such as virtual memory are handled differently and the primitives required to manipulate entities such as processes differ from those required to handle files.

Killian [4] describes a special file system (called */proc*) for the UNIX operating system which disguises active processes as files. Each "file" in */proc* corresponds to the address space of a running process. Singleton et. al. [9] introduce the concept of *extrafiles*. Implemented under UNIX, these act as a single model for the normal uses of both files and processes.

7. Conclusions

We have attempted to present a uniform interface to system resources that requires minimal knowledge by the user of the differences between those resources. The primitives described allow access to system resources without explicit knowledge of the type of resource accessed. Instead, the type of resource desired is specified by its behavior, based on an object which exhibits that behavior. Although this requires some initial knowledge on the part of the user (i.e., the desired behavior and an object known to exhibit that behavior), once an object has been created, it may be used in exactly the same manner as any other. Clearly, knowing the desired behavior of an object is necessary for its use. Short of requiring that the system recognize arbitrary object descriptions, we were unable to weaken the requirement that the user know of a currently existing object to act as a template. In spite of this, the amount of knowledge the user requires about resources in **GARD** is far less than that required in conventional operating systems.

8. Acknowledgements

We would like to thank Dr. Wlodzimierz Dobosiewicz for posing this problem to us, and for his suggestions and support. As well, we would like to thank Prakash Bettadapur for his contributions to the original paper on which this is based, and Ken Hruday for his numerous suggestions and critiques of the paper.

References

1. F. Corbató and V. Vyssotsky, "Introduction and Overview of the Multics System", *AFIPS Conference Proceedings*, Vol. 27, No. 1, 1965, pp. 185-196.
2. E. Gehringer, in *Capability Architectures and Small Objects*, UMI Research Press, Ann Arbor, Michigan, 1982.
3. A. Jones, "The Object Model: A Conceptual Tool for Structuring Software", *Lecture Notes in Computer Science*, Vol. 60, 1978,

pp. 8-16.

4. T. J. Killian, Processes as Files, *Usenix Conference, Summer 84*, Salt Lake City, 1984, pp. 203-207.

5. Henry M. Levy, in *Capability-Based Computer Systems*, Digital Press, Bedford, Mass., 1984.

6. E. I. Organick, in *A Programmer's View of the INTEL 432 System*, McGraw-Hill Book Company, New York, 1983.

7. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *Comm. ACM*, Vol. 17, No. 7, Jul 1974, pp. 365-375.

8. T. Rus, in *Data Structures and Operating Systems*, John Wiley & Sons, New York, NY, 1979.

9. P. Singleton, K. H. Bennett and O. P. Brereton, "A Single Model for Files and Processes", *ACM Operating Systems Review*, Vol. 20, No. 1, Jan 1986, pp. 12-18.

10. M. V. Wilkes and R. M. Needham, in *The Cambridge CAP Computer and Its Operating System*, P. J. Denning (ed.), North Holland, New York, 1979.