Esprit Project N° 6603 **OUVERTURE**

Applying Object Oriented Interfaces between the OM and PM

date:	93/06/10	*ref:*	**OU/TR-93-5**
category:	Report	*workpackage/task:*	WP1/T1.1
author(s):	Douglas Steel (USLE)	*distribution:*	partners
clearance:	Consortium	*status:*	FINAL
authorised by:	Nigel Colgan	*classification:*	INTERNAL

abstract: This document reports the initial analysis of applying OUVERTURE Interface Definition Language (OIDL) interfaces to the interaction of the Object Manager and Process Manager of the CHORUS/MiX microkernel based Operating System.

This document is written as part of the ESPRIT OUVERTURE project.

1. INTRODUCTION

This report covers the initial investigation into providing object oriented interfaces between the Object Manager (OM) and the Process Manager (PM). The OM and PM are Chorus Actors which implement some of the UNIX functionality in the CHORUS/MiX system. The current interfaces between these actors is only loosely defined. Even so the current interfaces are based on an object semantics for many operations. The purpose of this study is to build on these object semantics to provide an object-oriented interface between the PM and OM.

1.1 Overview

The approach used in this study is to:

- identify the current interactions between the OM and PM
- develop class definitions required for the interactions
- develop an interface definition based on these classes
- provide a few examples of the use of these interfaces

1.2 Scope

We do not intend to provide a complete definition of all interfaces between the OM and PM, instead we intend here to provide a set of examples of how this process should take place. Future work is intended to take this work to its logical conclusion (i.e. the definition and implementation of a completely object oriented interface between the OM and the PM). The interface definitions are in OIDL and are based on C++ classes. Eventual implementation will be based on the Chorus Object Oriented Layer (COOL). The addition of a well-defined interface would alleviate several problems with the current implementation, and would enable run time optimisations to be carried out.

1.3 Audience

This report is aimed at all those involved or interested in Task T1.1 *Object-based services for MiX* of Workpackage WP1 of the ESPRIT OUVERTURE project. Some knowledge of C++ and OIDL is assumed.

1.4 Reason for Reissue

This is the first issue.

1.5 Trademarks

The following trademarks are used and acknowledged in this document.

CHORUS	is a registered trademark of Chorus systèmes.
CHORUS/MiX	is a trademark of Chorus systèmes.
UNIX	is a registered trademark of UNIX System Laboratories, Inc.

1.6 Glossary

The following list defines acronyms and terms used in this document that may not be familiar to the readers.

Actor	An autonomous object defined by the CHORUS Nucleus, an Actor consists of an address space and a number of threads of control.
C	A systems programming language.
C++	An object-oriented language based on the syntax of C.
CEC	Commission of the European Communities. Promoter of ESPRIT programme.
Chorus	Chorus systèmes, OUVERTURE participant.
CHORUS/MiX	An object-oriented operating system built to run on top of CHORUS Nucleus.

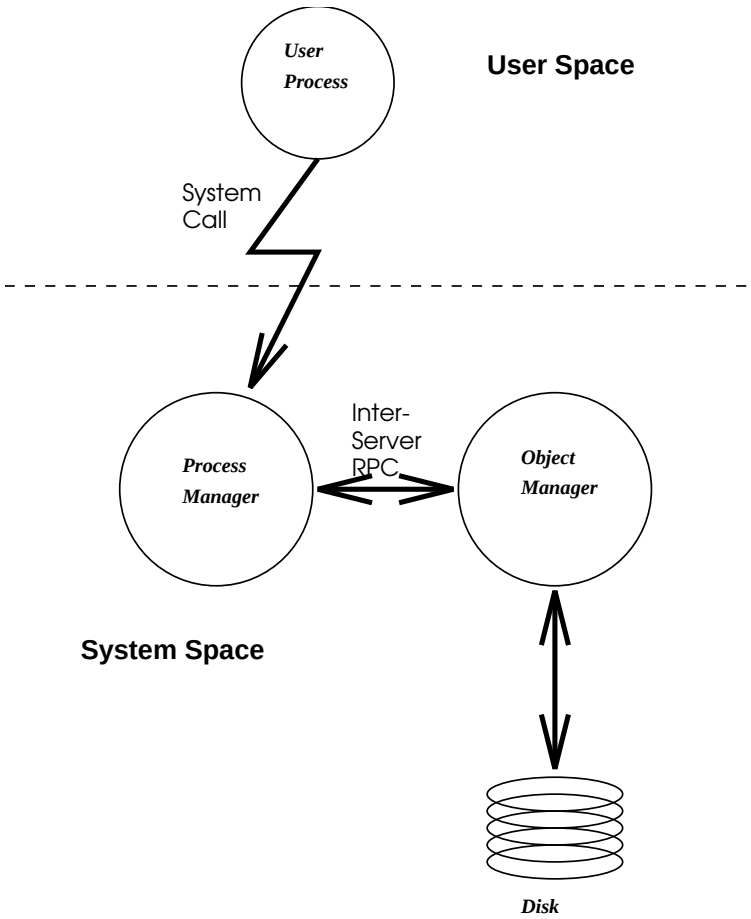
CHORUS Nucleus	An operating system kernel with microkernel architecture developed by Chorus systèmes.
COOL	The Chorus Object Oriented Layer.
COOL-L	COOL-Lite, a cut down version of COOL optimised for system server interactions.
COOL-UL	COOL-UltraLite, a cut down version of COOL optimised for system server interactions on the same site.
ESPRIT	European Strategic Programme for Research and Development of Information Technology.
IDL	Interface Definition Language.
Object	a unit of encapsulation consisting of data and the operations that may be performed on that data.
Object Manager	An Actor which looks after file objects, part of CHORUS/MiX.
OIDL	OUVERTURE Interface Definition Language.
OUVERTURE	ESPRIT project to progress microkernel-based open operating systems for the 90's.
process	A unit of execution in the UNIX system, consists of an address space and a single thread of execution.
Process Manager	An Actor which looks after the processes on the CHORUS/MiX system.
SNI	Siemens Nixdorf Informationssysteme AG, OUVERTURE participant.
Task	Part of the OUVERTURE Workpackage.
USL	UNIX System Laboratories Inc.
USLE	UNIX System Laboratories, Europe Ltd; subsidiary of USL. OUVERTURE participant.
Workpackage	Main part of the OUVERTURE project plan.
WP1	Distributed microkernel based SVR4. Workpackage 1 of OUVERTURE project.
WP2	Distributed microkernel technology. Workpackage 2 of OUVERTURE project.
WP3	Distributed embedded real-time systems. Workpackage 3 of OUVERTURE project.
WP4	Management and Liaison. Workpackage 4 of OUVERTURE project.

2. OM/PM INTERACTION

2.1 Approach

The current OM/PM interactions are defined in a rather ad hoc manner. Although all the arguments are passed in structures appropriate for the server call, all marshaling/demarshaling is performed manually in an irregular manner. For instance, some arguments are passed in the message body or annex depending on the circumstances. This is actually an optimisation used mainly for read/write. It does not currently seem possible to specify this arrangement in OIDL.

At other times, some arguments are passed and then never used (e.g. jc_params are never used in the OM - however they are used in the STM so will be kept in the interface). The diagram below shows the path of a typical system call which accesses a disk file system. The user process executes a system call which traps to the PM. The PM sends an interserver RPC to the OM which performs any disk access required, returning the result to the PM, and from there to the user process.



Compile time switches are used to determine the message passing mechanism (i.e. LWRPC or FWRPC), with the server needing to decide on the mechanism at compile time. The mechanism used is transparent to the client.

2.2 Types of Interaction

The interaction of PM and OM can be categorised into :

- initialisation

- named files
- open files
- file objects
- credentials caching
- token management
- file system locking
- `/proc`

Various messages are sent between the PM and OM on initialisation. These messages deal with operations such as the PM getting the root capability and the PM getting the amount of swap space. These operations are fairly simple, but are executed only once, they are therefore not considered to be of particular interest in this study.

Open files are a simple mapping of the system interface for file descriptor operations. An interface to an open file will be returned on an `open` operation on a named file.

Named files are those which are explicitly named by a pathname. The pathname may be relative or absolute. Named files referenced by absolute pathname (all those beginning with a ‘‘/’’) are managed by the OM referred to by the context’s root capability. All other named files are managed by the current directory capability.

File objects are the PM’s handle on to a cacheable open file’s vnode and is used for operations such as mmap, first open/last close, etc.

Credentials consist of the <uid/gid/euid/egid/suid/sgid/supplementary_groups/...> tuple and is used to determine the user’s access permissions to a file. In the initial implementation it is planned to leave credentials as they are currently implemented, they will be dealt with at a later stage.

Tokens are used to control the access to file length. They are passed back and forth between the PM and OM to enable one or the other to have exclusive access. In the initial implementation it is planned to leave these tokens as they are currently implemented, they will be dealt with at a later stage.

File system locks are implemented by the PM and exported as an interface to the OM. In the initial implementation it is planned to leave file system locks as they are currently implemented, they will be dealt with at a later stage.

The `/proc` filesystem is exported by the PM and is accessed through the OM. So the PM sends operations to the OM as if the `/proc` was a filesystem handled by the OM. The OM then sends the message back to the PM for it to handle the request.

In this paper I will only consider fully named and open files, and file objects. Credentials and token management will be touched upon (although will not be implemented initially).

2.3 Problems

Several problems arise from the current mechanism of server interaction:

- It is not always clear exactly which parameters are needed for each server call.
- Modifying the current interface is an involved process.
- Optimisations are difficult.

The addition of a well-defined interface would alleviate these problems, and would enable run time optimisations to be carried out.

3. DEFINING CLASSES FOR OM/PM INTERACTION

3.1 General Classifications

The major object classes that have been identified as arguments in OM/PM interaction are:

- Buffers
- Pathnames
- Open Files
- Named Files
- File Objects
- Credentials
- File System Context
- Tokens
- File System Locks

These are studied below. Note that any definitions presented here are only intended to give a flavour of a possible implementation and are thus incomplete.

In addition, the following major objects have been identified as receivers of requests :

- Open Files
- Named Files
- File Objects
- Credentials
- Tokens
- File System Locks

These will form the interfaces between the OM and PM and are discussed in the next section.

3.2 Buffers

A system call may pass down to the kernel a user specified buffer to be written to or read from. Data needs to be copied into/out-of these buffers. These buffers are passed as arguments between the PM and OM. The data can either be passed by value, in which case the IPC message body contains the data, or by reference, in which case a capability/address pair is sent as the reference. It is assumed here that the argument marshaling strategy will make this decision and take the appropriate actions. The class definition for a buffer is shown below.

```
class   Buffer {
                KnCap   cap;            // segment capability
                VmAddr  address;        // segment offset
                VmSize  size;           // current size of data
                VmSize  max;            // max. size of data
                int     flags;          // vm flags (eg. K_USERBODY)
public:
                        Buffer(const KnCap&,VmAddr=0,VmSize=0,int=0);
                        ~Buffer();
                VmSize  getSize() const;        // return size
                void    setSize(VmSize);        // new size
                VmAddr  getAddress() const;     // get a pointer to the data
                void    setAddress(VmAddr);     // new pointer
                KnCap   getCap() const;         // get a capability for data
                void    setCap();               // set capability for data
                int     isUserBuffer();         // return whether the buffer
                                                // is in user land
};
```

Implementation details are not discussed here. *Buffers* will tend to be passed as arguments to the `read` and `write` system calls. The PM only passes these on to the OM in the case of uncacheable files. The current implementation passes actual buffer data in the message body when the buffer size is small, and passes a reference when buffer size is large. The above definitions allow for a similar choice to be made by the PM's argument marshaling strategy (the PM always knows the size of the request).

3.3 Pathnames

Pathnames may be passed in a similar manner as buffer data in a message body, with one restriction: pathnames are always passed from the PM to the OM, and are never modified by the OM. A pathname has a system defined maximum size (in CHORUS/MiX V.4 Rel 2.0 it is 1024 characters). Pathnames are passed in all interactions where an open file context is not specified. It is assumed here that the argument marshaling strategy will only pass the number of characters specified by the size of the pathname.

```
class   Pathname {
                KnCap   cap;
                VmAddr  name;
                VmSize  size;
                int     flags;

public:
                Pathname (const KnCap&, VmAddr=0, VmSize=0, int=0);
                ~Pathname();
        VmAddr  name();
        void    newName(const VmAddr);
};
```

This is very similar to the definition of a *Buffer* above (a *Buffer* is really an extended pointer). It may be reasonable to inherit most of the behaviour form the *Buffer* class.

3.4 Named Files

These have been determined to be a receiver of interserver requests and will thus be described in the **INTERFACE DEFINITIONS** section.

3.5 Open Files

These will be grouped together and act as an interface definition. Since this class will be generated by the OIDL compiler it will not be discussed here. This is described in the **INTERFACE DEFINITIONS** section.

3.6 File Objects

File objects are the PM's handle on to a cacheable open file's vnode and are used for operations such as mmap, first open/last close, etc. Since this class will be generated by the OIDL compiler it will not be discussed here. This is described in the
INTERFACE DEFINITIONS section.

3.7 File System Context

In CHORUS/MiX the PM's file system context is made up of:

- the current directory
- the root directory
- the credentials object

This file system context is passed with every request which deals with pathname lookup. It is currently passed as part of the message header (msgHead).

```
class   FsContext {
        NamedFile       *current;       // interface for the current dir manager
        NamedFile       *root;          // interface for the root dir manager
        KnCap           *cred;          // user's credentials
                                        // this may become a Credentials
                                        // interface object
public:
                        FsContext(const NamedFile*,
                                const NamedFile *, const KnCap *);
                        ~FsContext();
        NamedFile       *getRoot();
        NamedFile       *getCurrent();
        KnCap           *getCurrent();
        void            setRoot(const NamedFile*);
        void            setCurrent(const NamedFile*);
        void            setCred(const KnCap*);
        // return an interface to the manager who manages
        // this Pathname either root or current
        NamedFile       *getManager(Pathname p&);
};
```

3.8 Tokens

The OM and PM pass tokens between each other, a PM can request a token from an OM, and an OM can recall a token from a PM. This will be used as an example of bi-directional access. One solution would be to implement a separate token manager which can respond to requests from both the OM and PM, however this still has the problem of needing to perform upcalls to one server or another. The solution used here is:

- PM exports a *Token* interface which has one method *recall()*.
- OM exports one operation *getToken()* as part of the *OpenFile* interface as an interface to its *Token*s

This is described in the **INTERFACE DEFINITIONS** section. This will not be implemented in the initial prototype.

3.9 Locks

File system locks are used in both the PM and OM, however it is always the PM that manages them, thus the PM will export an interface to its locks.

3.10 /proc

The process file system (/proc) is managed by the PM. However it is accessed just like any other file system, i.e. the PM sends its requests to the OM. The OM then send requests back to the PM's /proc handler for processing.

4. INTERFACE DEFINITIONS

The interface definitions for the OM/PM interface (shown below) are written in the OIDL (OUVERTURE Interface Definition Language) see [1]. All operations on named files will send messages to a *NamedFile* object which has a defined interface. Operations on open files will be sent to an *OpenFile* object. Only some simple operations will be defined in the interface here, some more complex examples are shown in the section entitled **CASE STUDIES - INTERFACES**.

4.1 Named File Interface

Named files are those which are explicitly named by a pathname. The pathname may be relative or absolute. Named files referenced by absolute pathname (all those beginning with a ‘‘/’’) are managed by the OM referred to by the context’s root directory. All other named files are managed by the OM referred to by the context’s current directory.

```
interface NamedFile{

        int     access (in Pathname, in FsContext, in int amode);
        int     chown (in Pathname, in FsContext, in uid_t, in gid_t);
        int     chmod (in Pathname, in FsContext, in mode_t);
        int     chroot (in Pathname, in FsContext, in KnUniqueId pmsite,
                                out NamedFile newroot, out dev_t newfsid);

                :
        < other definitions go here >
                :
};
```

4.2 Open File Interface

The OM will export an interface to open files. Currently this is done at the capability level, the OM maintains a capability for each open file and hands out these capabilities to the PM. The PM then refers to the open file by its capability. This general form of interaction can be preserved, although it should be encapsulated in a more specific open file object type rather than raw capabilities.

```
interface OpenFile {

        int     fchown(in uid_t, in gid_t);
        int     fchmod(in mode_t);

                :
        < other definitions go here >
                :
};
```

4.3 File Object Interface

File objects are the PM’s handle on to a cacheable open file’s vnode and are used for operations such as mmap, first open/last close, etc.

```
interface FileObject {

        int     vnrel(in KnUniqueId pmsite, in int nopen);
        int     getfd(in KnUniqueId pmsite, in KnCap cred, out OpenFile);
        int     mmap(in KnCap cred, in off_t, in size_t,
                        in int prot, in out flags);
//
//      If credentials are implemented as an interface these last two
//      may become :
//
//      int     getfd(in KnUniqueId pmsite, in Credentials cred, out OpenFile);
//      int     mmap(in Credentials cred, in off_t, in size_t,
//                      in int prot, in out flags);
//
        int     munmap(in KnUniqueId pmsite, in int nopen, in int nmap,
                        in ushort sync_seqno, in out flags);
                :
        < other definitions go here >
                :
};
```

4.4 Credentials

The PM currently passes the OM a capability for a credentials object. When the OM wishes to use such a credentials object it looks to see if it has a cached copy of the object and uses that. Otherwise the OM fetches a copy of the credentials object from the PM. Here a similar interface will be used. The argument that is passed with file operations will effectively be a reference to a credentials object (a capability), the OM will then have to fetch the actual credentials object from the PM, and store this in its cache. In the initial investigation we will keep the current credentials interface. However, if we keep the current style of interaction, the Credentials interface is very simple, consisting of just one operation:

```
interface Credentials {

        int     getCred(out cred_t cred);
}
```

4.5 Tokens

The interface that the PM exports for the tokens that it holds is very simple consisting of one method :

```
interface Token {
        int     recall();
}
```

The OM will export an interface to the PM to allow it to grab a token from the OM. Since *Token*s are related to open files this will be part of the *OpenFile* interface :

```
interface OpenFile {

                :
        < other definitions go here >
                :

        int     getToken(out Token);

                :
        < other definitions go here >
                :

}
```

4.6 File System Locks

The interface that the PM exports for file system locks is very simple, consisting of four methods. However, difficulties arise in this interface because of :

- Broadcast - some of the requests are broadcast to several PMs
- Some messages are not directed at a specific lock and are rather intended for the server which manages locks.

One solution to these problems would be, to define a lock manager which performs the requests for the client :

```
interface FsLockManager {

        int     buildLock(in FileObject, in int flags, in int nlocks,
                        in Buffer locks);
        int     undoLock(in FileObject, in FsLock);
        int     setLock(in FileObject, in FsLock);
        int     cleanLocks(in FileObject, in pid_t pid, in sysid_t system_id);

}
```

Further investigation is required here, and the initial implementation will preserve the current interface.

5. CASE STUDIES - INTERFACES

5.1 chown

The chown interface is relatively simple but is used here as an example of the *NamedFile* interface.

```
interface NamedFile{

                :
        < other definitions go here >
                :
        int     chown (in Pathname, in FsContext, in uid_t, in gid_t);
                :
        < other definitions go here >
                :
};
```

5.2 chroot

The chroot interface is a slightly more involved example of an *NamedFile* interface. One interesting point to note is that it returns as an out parameter a *NamedFile* interface. This will become the root portion of that processes *FsContext*.

```
interface NamedFile{

                :
        < other definitions go here >
                :
        int     chroot (in Pathname, in FsContext, in KnUniqueId pmsite,
                                out NamedFile newroot, out dev_t newfsid);
                :
        < other definitions go here >
                :
};
```

5.3 open

The open file protocol between the OM and PM takes as a parameter (amongst others) a *NamedFile* and returns an *OpenFile* object to the PM. The open file protocol is thus unique in the OM/PM interface in that it could belong to either the *OpenFile* interface or the *NamedFile* interface. Here it will be placed in the *NamedFile* Interface.

```
interface NamedFile {
                :
        < other definitions here >
                :
        int     open(in Pathname, in FsContext, in int creat_mode,
                        in int file_mod, in KnUniqueId pmsite, in int noctty,
                        in sysid_t system, in pid_t process, in pid_t sid,
                        in int nofiles, in KnKey pgorigin,
                        out OpenFile, out FileObject, out size_t size,
                        out int alloctty, out dev_t fsid, out dev_t tty_rdev,
                        out int fifo_blk);
                :
        < other definitions here >
                :
};
```

The open file protocol also return a *FileObject* and has a rather large number of parameters. It may be desirable to hide this interface with a higher layer which encapsulates these parameters. An alternative interface based on the miscellaneous `in` and`out` parameters being encapsulated in *OpenInParams* and *OpenOutParams* could be :

```
interface NamedFile {
                :
        < other definitions here >
                :
        int     open(in Pathname, in FsContext, in OpenInParams,
                        out OpenFile, out FileObject, out OpenOutParams);
                :
        < other definitions here >
                :

}
```

5.4 `read/write`

The read and write protocols are relatively straight forward and will therefore be treated together. It should be noted that ‘‘normal’’ (i.e. cached) reads and writes are not handled by this mechanism, rather they are handled by the virtual memory system. The read and write interfaces both rely on the *Buffer* implementation described above, and operate on open files. The interface is as follows:

```
interface OpenFile {
                :
        < other definitions here >
                :
        int     read(in Buffer, in off_t offset, in unsigned long ulimit
                        in KnCap actor, in jcparam, out int nbytes);
        int     write(in Buffer, in off_t offset, in unsigned long ulimit
                        in KnCap actor, in jcparam, out int nbytes,
                        out int signo);
                :
        < other definitions here >
                :
};
```

Note that since the class *Buffer* (shown earlier) is really just an extended reference, it is actually an `in` parameter to both the`read` and `write` operations.

5.5 `ioctl`

The ioctl interface is unusual in OM/PM interaction in that it has not only variable size arguments, but that it also has a variable type of arguments. One way of dealing with this is to have a simple single interface which then deals with each separate interface based on the first argument, the first argument is an operation code, this is similar to the current approach.

Another solution would be to have a separate interface for each operation code, this is both cleaner and more object oriented, but leads to a proliferation of interfaces. Another solution would be to have a set of interfaces for each different signature of arguments, this is a compromise between the other approaches, but has neither the simplicity of the first solution nor the elegance of the second. Shown below is a portion of the second solution. Interface:

```
interface OpenFile {
                :
        < other definitions here >
                :
        int     ioctl_fionread(out off_t offset);
        int     ioctl_fionbio(in int flag);
        int     ioctl_getalttbl(out stuct alt_info);
        int     ioctl_vconfig(in karg);
        int     ioctl_vremount();
        int     ioctl_vgetparms(out disk_params);
                :
        < other definitions here >
                :
};
```

Only a few interfaces are shown above, but it can be seen how these are already proliferating. This approach also makes it difficult to add new ioctls or port existing drivers. The compromise solution also has these difficulties.

The first solution presents a much simpler interface:

```
interface OpenFile {
                :
        < other definitions here >
                :
        int     ioctl(in int cmd, in Buffer arg, in uid_t uid, in pids,
                        in noctty, in KnCap actor);
                :
        < other definitions here >
                :
};
```

This simple interface, however, moves the complexity into the rest of the PM/OM/STM code, which has to then select the appropriate argument type based on the command. A large number of ioctls are directed towards the STM and adopting any other approach would result in having rather different interfaces between the PM/OM and the PM/STM. Since this approach is very similar to the current approach this will be the method used in the first prototype implementation. Future implementors would be advised to look at this issue again.

5.6 fcntl

The `fcntl` is similar to the`ioctl` interface, except that `fcntl` only relates to ‘‘normal’’ files, and it has a much more well defined interface with only a few commands. It would therefore be possible to define the fcntl interface by specifying a separate interface for each command :

```
interface OpenFile {
                :
        < other definitions here >
                :
        int     fcntl_dupfd(in int arg);
        int     fcntl_getfd(in int flags);
        int     fcntl_setfd(in int fd);
        int     fcntl_getfl();
        int     fcntl_setfl(in int fl);
        int     fcntl_getlk(out struct flock);
        int     fcntl_setlk(in struct flock);
        int     fcntl_setlkw(in struct flock);
                :
        < other definitions here >
                :
};
```

Although the above interface is relatively clean and straightforward it would involve re-writing portions of the OM not related to the interface. For this reason, the initial implementation will take a similar approach to that used in the `ioctl` interface above.

```
interface OpenFile {
                :
        < other definitions here >
                :
        int     fcntl(in int cmd, in Buffer arg);
                :
        < other definitions here >
                :
};
```

5.7 mmap

The `mmap` operation is shown here as an example of the *FileObject* interface. It is relatively simple, passing a credentials object, and the description of the address space to be mapped.

```
interface FileObject {

                :
        < other definitions go here >
                :

        int     mmap(in KnCap cred, in off_t, in size_t,
                        in int prot, in out flags);
                :
        < other definitions go here >
                :
};
```

6. CONCLUSIONS

The above study is incomplete and unimplemented. It is intended only to gauge the feasibility of applying OIDL interfaces to OM/PM interaction and to provide an initial classification of these interactions. The OM/PM interface will be an evolving target and hence this study is based only on a snapshot of the current interface.

I feel that the study presented in this report shows that the implementation of OIDL interfaces between the OM and PM are not only feasible, but also desirable. It is hoped that the OIDL compiler and COOL-Lite system will allow a well defined object-oriented interface, such as that envisioned here, to be implemented efficiently.

The next step in this process will be to further define and implement a prototype using the interfaces described in this report. This will be a multi phase project with each phase gaining insight from the previous. Eventually this work will be combined with the work under progress on the PM/KM/IPCM interfaces and future work on the STM interface.

Appendix A — CASE STUDIES - USAGE

In this section, several examples will be used to show how the PM side of the OM/PM interactions will be modified with the application of OIDL interfaces. All code shown here is for example purposes and is incomplete. Code is derived from the files `pmFile.cxx` and `pmFlObj.cxx`.

A.1 `chown`

Current implementation:

```
int     TUxProcFile::Chown      (tPmUThread* UT)
{
```

>>> stuff deleted <<<

```
Requestp->service       = S_UN_CHOWN;
Requestp->new_uid       = newuid;
Requestp->new_gid       = newgid;
SetMsgCtx(&Requestp->msgHead.msg_ctx);

tPmProc *P = UT->MyProc();
tPmCred *cr = P->Cred.credp;

if (P->Cred.IsSuser()); // Scott -- use the side effect of this line
                        //          to pass the BCS accounting test

Requestp->msgHead.credCap = cr->CredCap;

ReqMsg.bodyAddr  = (VmAddr) Requestp;
ReqMsg.annexAddr = 0;
ReqMsg.bodySize  = index;
ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL ;

MsgRep.flags     = 0;
MsgRep.bodyAddr  = (VmAddr) &Reply;
MsgRep.bodySize  = sizeof(TUnChownReply);
MsgRep.annexAddr = 0;

        // Send the message to the Object manager, wait for the reply
ReturnCode = ServiceCall(UT, K_DEFAULTPORT, Om_Dest(Requestp->path),
                        &ReqMsg, &MsgRep, 0);
if (ReturnCode)
        return ReturnCode;
return Reply.diag;
}
```

New implementation:

```
int     TUxProcFile::Chown      (tPmUThread* UT)
{

FsContext context;
NamedFile *manager;
```

>>> stuff deleted <<<

```
        tPmProc *P = UT->MyProc();
        tPmCred *cr = P->Cred.credp;

        if (P->Cred.IsSuser()); // Scott -- use the side effect of this line
                                //          to pass the BCS accounting test

        context.setCred(cr->credCap);
        manager = context.getManager(path);

        // Send the message to the Object manager, wait for the reply

        ReturnValue = manager->chown(path,context,newuid,newgid);
        return ReturnValue;
}
```

A.2 chroot

Current implementation:

```
int     TUxProcFile::Chroot     (tPmUThread* UT)
{
```

>>> stuff deleted <<<

```
        Requestp->service       = S_UN_CHROOT;
        SetMsgCtx(&Requestp->msgHead.msg_ctx);

        tPmCred *cr = P->Cred.credp;

        Requestp->msgHead.credCap = cr->CredCap;

        Requestp->pmsite = PmRequestPortUI;

        ReqMsg.bodyAddr  = (VmAddr) Requestp;
        ReqMsg.annexAddr = 0;
        ReqMsg.bodySize  = index;
        ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL ;

        MsgRep.flags     = 0;
        MsgRep.bodyAddr  = (VmAddr) &Reply;
        MsgRep.bodySize  = sizeof(TUnChrootReply);
        MsgRep.annexAddr = 0;

        ReturnCode = ServiceCall(UT, K_DEFAULTPORT, Om_Dest(Requestp->path),
                                 &ReqMsg, &MsgRep, 0);
        if (ReturnCode) {
                return ReturnCode;
        }
        if (Reply.diag) {
                return (Reply.diag);
        }

        if ((fobjp = SysFObjList.Add(&(Reply.new_root), 0, 1, IS_DIRECTORY,
                                 Reply.new_fsid)) == NULL) {
                // Should undo the message sent to the OM
                printf ("PM: chroot should undo request in the OM0);
                return ENFILE;
        }

        Frootobj->DirRel();
        Frootdir = Reply.new_root;
        Frootobj = fobjp;
        return 0;
}
```

New implementation:

```
int     TUxProcFile::Chroot     (tPmUThread* UT)
{
FsContext context;
NamedFile *manager;

 >>> stuff deleted <<<

        tPmProc *P = UT->MyProc();
        tPmCred *cr = P->Cred.credp;

        context.setCred(cr->credCap);
        manager = context.getManager(path);

        ReturnCode = manager->chroot(path,context,pmsite,new_root,new_fsid);

        if(ReturnCode) return ReturnCode;

        if ((fobjp = SysFObjList.Add(&new_root, 0, 1, IS_DIRECTORY,
                                new_fsid)) == NULL) {
                // Should undo the message sent to the OM
                printf ("PM: chroot should undo request in the OM0);
                return ENFILE;
        }

        Frootobj->DirRel();
        Frootdir = new_root;
        Frootobj = fobjp;
        return 0;
}
```

A.3 open

Current implementation:

```
int     TUxProcFile::OpenFile(tPmUThread *UT, char *name,
                int rwmode, int crmode,
                TPmFObj **fobjpp, TPmOpenFile **fpp, int flag,
                int PortLi, unsigned long *keytail)
{
```

>>> stuff deleted <<<

```
        tPmCred *cr = P->Cred.credp;
        Requestp->msgHead.credCap = cr->CredCap;

        Requestp->service       = S_UN_OPEN;
        Requestp->filemode      = rwmode;
        Requestp->creatmode     = crmode & ~Fumask;
        Requestp->pmsite        = PmRequestPortUI;
        Requestp->system_id     = pmSystemId;
        Requestp->process_id    = mypid;
        // we don't care about Requestp->nofiles and Requestp->noctty

        ReqMsg.bodyAddr  = (VmAddr) Requestp;
        ReqMsg.annexAddr = 0;
        ReqMsg.bodySize  = index;
        ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL;

        MsgRep.bodyAddr  = (VmAddr) &Reply;
        MsgRep.annexAddr = 0;
        MsgRep.bodySize  = sizeof(TUnOpenReply);
        MsgRep.flags     = 0;

        ReturnCode = ServiceCall(UT, PortLi, Om_Dest(name), &ReqMsg, &MsgRep, 0);
        if (ReturnCode) {
```

>>> stuff deleted <<<

```
        }
}
```

New implementation:

```
int     TUxProcFile::OpenFile(tPmUThread *UT, char *name,
                int rwmode, int crmode,
                TPmFObj **fobjpp, TPmOpenFile **fpp, int flag,
                int PortLi, unsigned long *keytail)
{
FsContext context;
NamedFile *manager;
OpenOutParams   out;

 >>> stuff deleted <<<

        tPmProc *P = UT->MyProc();
        tPmCred *cr = P->Cred.credp;

        context.setCred(cr->credCap);
        manager = context.getManager(path);
        OpenInParams    in(pmsite, 0, pmSystemId, mypid, 0, 0);

        // Send the message to the Object manager, wait for the reply

        ReturnCode = manager->open(path,context,crmode&Fumask,rwmode,in,
                        ofp, fobj, out);

 >>> stuff deleted <<<

}
```

A.4 fcntl

Current implementation:

```
int     TPmOpenFile::Fcntl(tPmUThread *UT, TUxProcFile *, tPmProc *P,
                        int Command, int Arg, int isBCS, long &rval1)
{
```

>>> stuff deleted <<<

```
        Requestp->service       = S_UN_FCNTL;
        Requestp->open_file     = f_FileCap;
        Requestp->cmd           = Command;
        Requestp->flag          = f_flag;
        relmutex();

        ReqMsg.bodyAddr  = (VmAddr) Requestp;
        ReqMsg.annexAddr = 0;
        ReqMsg.bodySize  = index;
        ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL;

        MsgRep.flags     = 0;
        MsgRep.bodyAddr  = (VmAddr) &Reply;
        MsgRep.bodySize  = sizeof(TUnFcntlReply);
        MsgRep.annexAddr = 0;

        if (ReturnCode = ServiceCall(UT, K_DEFAULTPORT, &(f_FileCap.ui),
                                &ReqMsg, &MsgRep, P))
                return ReturnCode;

        if (Reply.diag)
                return (Reply.diag);

        // we reach here when no error returned from the server
        switch (Command) {
```

>>> stuff deleted <<<

```
        }
}
```

New implementation:

```
int     TPmOpenFile::Fcntl(tPmUThread *UT, TUxProcFile *, tPmProc *P,
                        int Command, int Arg, int isBCS, long &rval1)
{
Buffer  buffer;

 >>> stuff deleted <<<

        relmutex();

        if (ReturnCode = open_file->fcntl(Command,buffer,f_flag);
                return ReturnCode;

        // we reach here when no error returned from the server
        switch (Command) {
         >>> stuff deleted <<<
        }
}
```

A.5 ioctl

Current implementation:

```
int     TPmOpenFile::Ioctl (tPmUThread* UT, TUxProcFile * Pfile, int Cmd,
                int Arg, unsigned Ctty, long &rval1, int inKernel, int isBCS)
{
>>> stuff deleted <<<

        Requestp = (TUnIoctlRequest *)(UT->SysCall.GetComAddress());
        Requestp->cmd           = Cmd;
        Requestp->arg           = Arg;
        Requestp->uid           = P->Cred.GetUid();
        Requestp->pids.pid      = P->UI.GetPid();
#ifndef USER_ACTOR
        Requestp->pids.pgid     = P->UI.GetPgid();
        Requestp->pids.sid      = P->UI.GetSid();
#else
        Requestp->pids.pgid     = 1;    // JJJ -- TBD
        Requestp->pids.sid      = 1;    // JJJ -- TBD
#endif
        Requestp->pids.stopreads        = UT->Sig.SigEnabled(SIGTTIN);
        Requestp->pids.stopwrites       = UT->Sig.SigEnabled(SIGTTOU);
        Requestp->noctty = 0;

>>> stuff deleted <<<

        Requestp->service       = S_UN_IOCTL;
        Requestp->open_file     = f_FileCap;
        if (inKernel)
                Requestp->actor_cap = PmCap;    // ioctl initiated by socket
        else if (actorSelf(&(Requestp->actor_cap))) {
                return EFAULT;
        }

        ReqMsg.bodyAddr  = (VmAddr) Requestp;
        ReqMsg.annexAddr = 0;
        ReqMsg.bodySize  = index;
        ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL;

        MsgRep.flags     = 0;
        MsgRep.bodyAddr  = (VmAddr) &Reply;
        MsgRep.annexAddr = 0;
        MsgRep.bodySize  = sizeof(TUnIoctlReply);

        ReturnCode = ServiceCall(UT, K_DEFAULTPORT, &f_FileCap.ui,
                                        &ReqMsg, &MsgRep, P);

        switch(Cmd) {
>>> stuff deleted <<<
        }
        return 0;
}
```

New implementation:

```
int     TPmOpenFile::Ioctl (tPmUThread* UT, TUxProcFile * Pfile, int Cmd,
                int Arg, unsigned Ctty, long &rval1, int inKernel, int isBCS)
{

 >>> stuff deleted <<<

        uid             = P->Cred.GetUid();
        pids.pid        = P->UI.GetPid();
        pids.stopreads  = UT->Sig.SigEnabled(SIGTTIN);
        pids.stopwrites = UT->Sig.SigEnabled(SIGTTOU);
        noctty = 0;

        if (inKernel)
                actor_cap = PmCap;    // ioctl initiated by socket
        else if (actorSelf(&actor_cap)) {
                return EFAULT;
        }

        ReturnCode = open_file->ioctl(Cmd,Arg,uid,pids,noctty,actor,pgorigin,
                                result,file,alloctty,tty_rdev);

        switch(Cmd) {
        >>> stuff deleted <<<
        }
        return 0;
}
```

A.6 mmap

Current implementation:

```
int TPmFObj::Mmap(tPmUThread *UT, caddr_t Addr, size_t Len, int Prot, int Flags,
                        off_t Offset, long &rval1, tPmCred *crp, int maxprot,
                        size_t objLen )
{
 >>> stuff deleted <<<

        getmutex();
        if (Fo_pages) {
                sendMsg=0;
        } else {
                sendMsg=1;
        }
        Fo_pages += (((unsigned)(Len + PAGEOFFSET)) >> PAGESHIFT);
        relmutex();
        if (sendMsg) {

                Requestp->service       = S_UN_MMAP;
                Requestp->obj_cap       = Fo_cap;
                Requestp->off           = Offset;
                Requestp->len           = Len;
                Requestp->prot          = Prot;
                Requestp->flag          = Flags;
                Requestp->credCap       = crp->CredCap;

                ReqMsg.bodyAddr  = (VmAddr) Requestp;
                ReqMsg.annexAddr = 0;
                ReqMsg.bodySize  = index;
                ReqMsg.flags     = index < OmMaxCopy ? 0 : K_MOVEAL;

                MsgRep.bodyAddr  = (VmAddr) &Reply;
                MsgRep.annexAddr = 0;
                MsgRep.bodySize  = sizeof(TUnMmapReply);
                MsgRep.flags     = 0;

                if (ReturnCode = ServiceCall(UT, K_DEFAULTPORT, &(Fo_cap.ui),
                                        &ReqMsg, &MsgRep, 0)) {
                        Munmap (Len);
                        return ReturnCode;
                }
                if (Reply.diag) {
                        Munmap(Len);
                        return (Reply.diag);
                }
        } /* sendMsg */


         >>> stuff deleted <<<

        return 0;
}
```

New implementation:

```
int TPmFObj::Mmap(tPmUThread *UT, caddr_t Addr, size_t Len, int Prot, int Flags,
                        off_t Offset, long &rval1, tPmCred *crp, int maxprot,
                        size_t objLen )
{
 >>> stuff deleted <<<

        getmutex();
        if (Fo_pages) {
                sendMsg=0;
        } else {
                sendMsg=1;
        }
        Fo_pages += (((unsigned)(Len + PAGEOFFSET)) >> PAGESHIFT);
        relmutex();
        if (sendMsg) {

                if (ReturnCode = fobj->mmap(crp->CredCap,Offset,
                                        Len,Prot,Flags)) {
                        Munmap (Len);
                        return ReturnCode;
                }
        } /* sendMsg */

         >>> stuff deleted <<<

        return 0;
}
```

A.7 write

Current implementation:

```
int     TPmOpenFile::Write(tPmUThread *UT, tPmProc *P, TUnRw *CallParam,
                        long &rval1, int prw, int inKernel, TPmSoHashTbl **
#ifdef SOCKET
                                                        Fsockptr
#endif
)
{
```

>>> stuff deleted <<<<

```
        if (cacheable) {
```

>>> stuff deleted <<<<

```
        } else { // for all non cacheable files
```

>>> socket stuff deleted <<<<

```
                Request.service         = (prw) ? S_UN_PWRITE : S_UN_WRITE;
                Request.open_file       = f_FileCap;
                Request.address         = (int)UserBuffer;
                Request.nbytes          = mycount;
                Request.offset          = (prw) ? CallParam->Offset : 0;
                Request.ulimit          = myulimit;
                Request.pids.pid        = mypid;
#ifndef USER_ACTOR
                Request.pids.pgid       = P->UI.GetPgid();
                Request.pids.sid        = P->UI.GetSid();
#else
                Request.pids.pgid       = 1;    // JJJ -- TBD
                Request.pids.sid        = 1;    // JJJ -- TBD
#endif
                Request.pids.stopreads          = UT->Sig.SigEnabled(SIGTTIN);
                Request.pids.stopwrites         = UT->Sig.SigEnabled(SIGTTOU);

                if (inKernel)
                        Request.actor_cap = PmCap;
                else
                        actorSelf(&(Request.actor_cap));

                if (mycount <= pmPageSize) {
                        // use request msg body to transfer data
                        ReqMsg.bodyAddr = (VmAddr)UserBuffer;
                        ReqMsg.bodySize = mycount;
                } else {
                        // server should use vmCopy to transfer data
                        ReqMsg.bodyAddr = 0;
                        ReqMsg.bodySize = 0;
                }
                ReqMsg.flags     =  UT->SysCall.CallerIsUser() ? K_USERBODY : 0;
                ReqMsg.annexAddr = (VmAddr) &Request;
```

```
                MsgRep.flags     = 0;
                MsgRep.bodySize  = 0;
                MsgRep.annexAddr = (VmAddr) &Reply;

                if (ReturnCode = ServiceCall(UT, K_DEFAULTPORT, &(f_FileCap.ui),
                                             &ReqMsg, &MsgRep, P))
                        return ReturnCode;
```

>>> stuff deleted <<<

```
                return 0;
        }
}
```

New implementation:

```
int     TPmOpenFile::Write(tPmUThread *UT, tPmProc *P, TUnRw *CallParam,
                        long &rval1, int prw, int inKernel, TPmSoHashTbl **
#ifdef SOCKET
                                                              Fsockptr
#endif
)
{
```

>>> stuff deleted <<<<

```
        if (cacheable) {
```

>>> stuff deleted <<<<

```
        } else { // for all non cacheable files
```

>>> socket stuff deleted <<<<

```
                Buffer b(UserCap,(VmAddr)UserBuffer,mycount);
                pids.pid        = mypid;
#ifndef USER_ACTOR
                pids.pgid       = P->UI.GetPgid();
                pids.sid        = P->UI.GetSid();
#else
                pids.pgid       = 1;    // JJJ -- TBD
                pids.sid        = 1;    // JJJ -- TBD
#endif
                pids.stopreads          = UT->Sig.SigEnabled(SIGTTIN);
                pids.stopwrites         = UT->Sig.SigEnabled(SIGTTOU);

                if (inKernel)
                        actor_cap = PmCap;
                else
                        actorSelf(&(actor_cap));

                if (ReturnCode = open_file->write(b,offset,myulimit,
                                                  actor_cap,pids,nbytes,signo))
                        return ReturnCode;
```

>>> stuff deleted <<<

```
                return 0;
        }
}
```

CONTENTS

1. INTRODUCTION 1
   1.1 Overview 1
   1.2 Scope 1
   1.3 Audience 1
   1.4 Reason for Reissue 1
   1.5 Trademarks 1
   1.6 Glossary 1
2. OM/PM INTERACTION 3
   2.1 Approach 3
   2.2 Types of Interaction 3
   2.3 Problems 4
3. DEFINING CLASSES FOR OM/PM INTERACTION 5
   3.1 General Classifications 5
   3.2 Buffers 5
   3.3 Pathnames 6
   3.4 Named Files 6
   3.5 Open Files 6
   3.6 File Objects 7
   3.7 File System Context 7
   3.8 Tokens 7
   3.9 Locks 7
   3.10 `/proc` 8
4. INTERFACE DEFINITIONS 9
   4.1 Named File Interface 9
   4.2 Open File Interface 9
   4.3 File Object Interface 9
   4.4 Credentials 10
   4.5 Tokens 10
   4.6 File System Locks 11
5. CASE STUDIES - INTERFACES 12
   5.1 `chown` 12
   5.2 `chroot` 12
   5.3 `open` 12
   5.4 `read/write` 13
   5.5 `ioctl` 14

5.6 `fcntl` 15
5.7 `mmap` 15
6. CONCLUSIONS 16
Appendix A — CASE STUDIES - USAGE 17
A.1 `chown` 17
A.2 `chroot` 19
A.3 `open` 21
A.4 `fcntl` 23
A.5 `ioctl` 25
A.6 `mmap` 27
A.7 `write` 29