BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-94-M16

"MOM: A Memory Object Manager for BOSS"

by

Vincent C. Rubino

This research project by Vincent C. Rubino is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

May 1994

Date: 5/19/94

Stanley B. Zdonik

Stanley B. Zdonik

MOM:
A Memory Object Manager for BOSS

Vincent C. Rubino
Department of Computer Science
Brown University

May 19, 1994

MOM provides BOSS with a subsystem to manage persistent and volatile memory using memory mapping to move data between volatile and stable memory.

1 Introduction

Memory management is critical in an object store. The MOM subsystem was created to fill this space. What differentiates the current MOM from the previous MOM designs is that it uses memory mapping rather than read and write system calls. The choice pushes the details of system calls down a level while retaining control over replacement. The MOM architecture consists of two classes, Memory Objects (MOs) and Memory Object Managers (MOMs). The MOs reference desired volatile space while the MOM manages those MOs.

2 Architecture

The point of MOM is to provide storage classes with persistent and volatile storage. MOM manages a number of stable stores and an area of volatile memory as shown in Figure 1. When requested, MOM maps data from the stable store into volatile memory where it can be manipulated. Changes made to this volatile area will persist into stable memory through the process of unmapping and synchronization. Once the storage class is done with the MO, it will be unmapped at some later time.

There is a smoall difference between memory management at the server and client MOM. The server MOM will never remove a MO from stable storage unless told to do so by the storage class. The client MOM can delete MOs from stable storage once they are unpinned. Stable replacement of MOs from a server store would be inconsistent with the mission of a server as persistent storage.

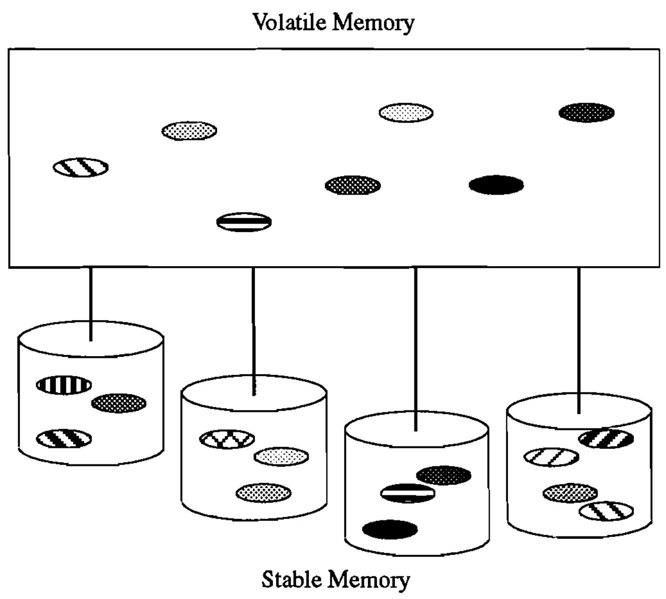

Figure 1: An illustration of mapping between stable and volatile memory.

For example, consider a server MOM controlled by a storage class. The storage class has a great need for memory, but does not need all of it all the time. It creates MOs, uses the memory and signals the MO when the data is not needed in volatile space. At any time, the storage class can ask for the MO to be moved back into volatile space for manipulation. MOM can move the MO out of volatile memory during the time it is no longer needed.

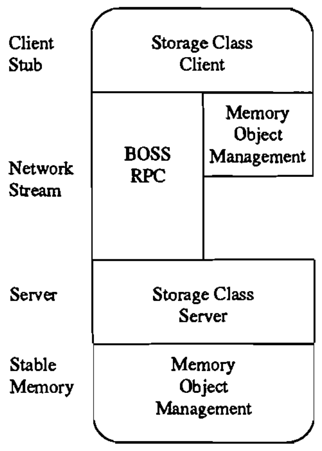

Figure 2: A layered view of a storage class.

A more complicated example is shown in Figure 2 which reflects the usage of BOSS. Each MOM manages its own stable store. The two MOMs function as in the above example, however, the top MOM is a client site which can remove its MOs from stable space. When this occurs, the client storage class might want a MO which has been removed from stable storage. In this case, the client storage class must get the MO from the server storage class. The client makes an RPC call to the server who asks its MOM to move the MO into volatile space where it can be copied into the given buffer. The buffer is then copied by the client storage class into a new MO at the client MOM.

3 Interface

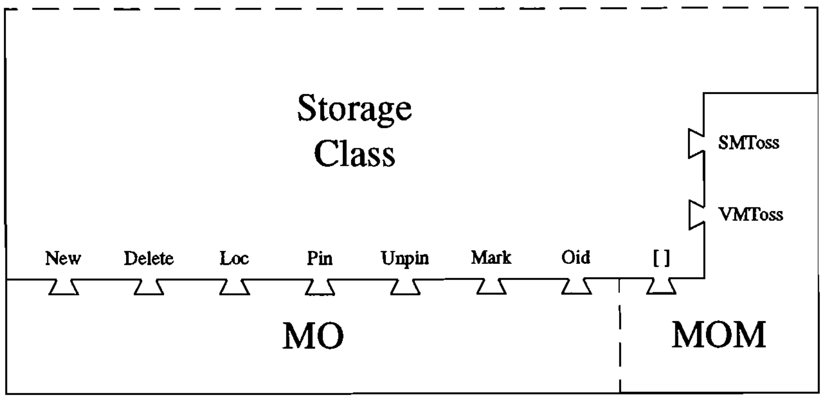

Figure 3: A view of the MOM storage class interface.

Calling these classes are a storage class which desires some volatile memory to be persistent. The interface between the three is shown in Figure 3.

A MO provides several access functions to the storage class. The primary functions include creation, deletion and location. Creating a MO provides the storage class with immediate access to volatile space. Deleting the MO will remove it from both volatile memory and from the stable store. The loc call provides the caller with a pointer to the volatile memory location secured by the MO. With these three calls, the storage class retains complete control over the MOM.

To enable the replacement policy of MOM the storage space is provided with three additional calls: pin, unpin and mark. To move a MO into volatile memory and keep it there, the storage class has the pin command. As with loc, a pointer accessing the volatile memory location is returned by the call. When a MO is no longer needed in volatile memory, but still might be needed, the unpin command is called. If the storage class wishes to give a MO a higher chance of remaining in memory once unpinned, the MO should be marked periodically. This call raises the priority of the MO with regards to replacement. Again, it should be noted that client MOMs have a policy enabling stable replacement of unpinned MOs.

The following interface tools provide a storage class with a bookkeeping mechanism to manage MOs solely via OIDs. There is a one for one relationship between MOs and OIDs. The OID of a MO can be both obtained and changed by calls to its MO. Furthermore, MOM keeps track of its MOs in terms of the OID. Thus, the storage class can obtain a pointer to a MO given the OID and the associative operator ([]) of MOM.

The storage class is also provided with a mechanism by which it is notified when a MO is moved out of memory. The storage class is expected to provide vmtoss and smtoss calls which the MOM will make whenever it moves a MO from memory. When a MO is removed from volatile memory, vmtoss is called. Similarly, smtoss is called whenever a MO is removed from stable memory. These callbacks serve as a notification to the storage class of an already removed

MO.

4 Implementation Issues

The interface of MOM hides a number of implementation issues from the storage class. To understand completely what is happening within MOM it is useful to look at these issues. They include: mapping, replacement, recovery and expansion.

4.1 Mapping

Rather than relying on direct read and write system calls, MOM makes use of mapping of stable memory into volatile space. The calls used are the mmap(), munmap() and msync() UNIX system commands.

Mmap(), munmap() and msync() provide a simple interface for mapping pages of stable memory into volatile memory. Mmap() maps a specified stable location into the process's address space. Once pages are mapped into volatile memory, the user can manipulate those pages as any other type of memory with the added assurance that changes will persist in stable memory. Munmap() closes the mapping opened through mmap while msync() asynchronously synchronizes volatile with stable memory.

These operations are used to forego the need for double buffering. The bookkeeping of reading and writing is pushed down a level while the control over replacement policies is held by MOM. Furthermore, MOM control makes mapping and unmapping invisible to the storage classes. Thus, the higher packages can just deal with MOs and OIDs and not have to worry about addresses, offsets and file descriptors.

4.2 Replacement

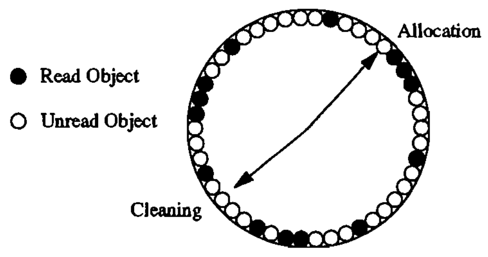

Figure 4: An illustration of the replacement clock hands.

Replacement is a complex issue that is handled similarly, but with some differences by the volatile and stable memory managers. The general policy for replacement is least recently used. It is simple to implement and is relatively quick.

The policy is effected through the movement of two pointers through the appropriate list of MOs. The cleaning pointer marks MOs as unread while the allocation pointer looks for unread MOs. MOs found by the allocation pointer are removed from their memory state unless prevented by some policy such as being pinned or being at a server MOM. These two pointers are illustrated in Figure 4.

MOs are deallocated by volatile replacement only when they are found to be unread and unpinned. Pinned MOs can be removed from volatile memory only when they are deleted. When MOs are deallocated, the vmtoss callback is made to notify the storage class.

Stable replacement can only occur on a client MOM site. MOs can be replaced in stable memory when they are found to be unread and are unpinned. The method of deallocation deletes the MO and then notifies the storage class via smtoss of the event.

4.3 Recovery

Most recovery issues in BOSS are handled outside the MOM package, but there are some basic things that need to be brought back on line when a site comes up. Namely, the MOM needs to be reloaded, repopulated and readied for action. To this end, anything that changes before a site goes down needs to be persistent.

The MOM has a small persistent file. This file holds irreplaceable information. Total size of volatile memory, client/server site status and pointers to managed stable stores are all in this file. Changes to this file are shadowed as changes are infrequent and the information in the file rarely referenced.

The call to make a new MOM will first check if a MOM file already exists. If it does, it recovers that file, repopulates MOM by following the pointers therein, and readies MOM for action. If a MOM file does not exist, a new MOM and a new MOM file are created.

Once the MOM file is safely loaded, the pointers are followed to recover the stable stores where the MO space and headers are actually stored. These stores are mapped into memory using mmap() as outlined above.

4.4 Expansion

Clearly, more stable memory will be needed at some point or another by servers and possibly by clients. To take care of the problem of expandability, two expansion commands are provided to the user. The two commands are createExtent() and removeExtent() reflecting the fact that the stable store class is called the extent.

CreateExtent() is the command called when the user wishes to add a large chunk of stable memory to the MOM space. Calls to this command will be infrequent at best as it will create files in the hundreds of megs and gigs. The call will initially update the MOM file so that in the event of failure later in the call, stable space will be added on recovery. In the event that creation is impossible rather than not completed, the reference to the new stable store will be removed from the MOM file.

RemoveExtent() is the other command provided to the user for dealing with expansion of MOM space. In this case, the space provided by an extent will be removed. As the user cannot know where MOs actually reside, this command should never be called from a server site. RemoveExtent() will merely remove the path from the MOM file (not deleting the extent file), so the stable space can be recovered at a later time by a call of createExtent() with the appropriate path.

5 Implementation Classes

Three classes are used in the implementation of the MOM subsystem. They are the MO, the MOM and the extent. The MO stores the information related to memory objects; the MOM manages MOs, stable memory stores and movement between stable and volatile memory; and the extent manages MOs use of stable memory.

5.1 Memory Objects

The implementation of the MO is straightforward. The MO stores data related to its state and provides hooks to the other two classes to access this information. In addition, it provides the public functions in the interface section.

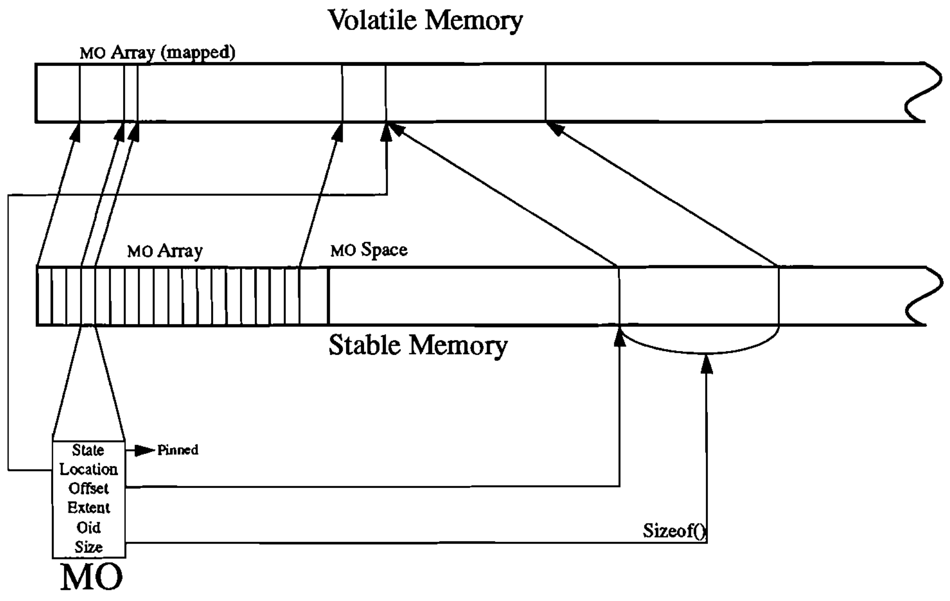

Figure 5: The relationship of MO with volatile and stable memory.

As a MO is persistent, the data of each instance is a chunk of stable memory mapped into volatile memory. To obtain this memory, the new operation is overloaded to obtain stable memory from the MOM. In addition to the instance space, there is a block of memory which is also allocated by the new operation. This block is the memory manipulatable by the storage class. An instance of a MO in relation to its stable space is shown in Figure 5.

The public functions of the MO check the MOs state and call MOM functions to enact major changes. For example, the MO changes its state values, but calls MOM to be mapped into volatile memory.

5.2 Memory Object Manager

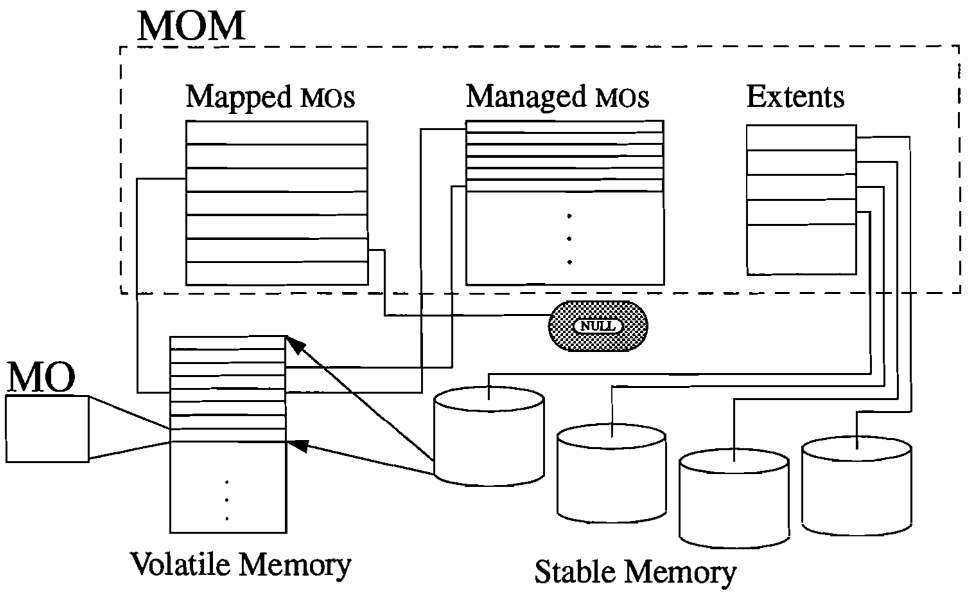

Figure 6: The structure of MOM with relation to memory.

MOMs implementation focuses on the management of MOs, management of stable stores where the MOs reside, and the management of volatile memory. An inkling to the complexity of this job can be seen in Figure 6. It provides a number of hooks for the other two classes to enact its managerial roles and update it on changes. Furthermore, it provides a public indexing hook returning the MO associated with a given OID.

The management of MOs consists of dealing with their construction and destruction, changes to their OID and requests to move the MO into volatile memory. When asked for stable space for a new MO, MOM asks its extents for a MO space until one is found or no space is found. In addition, the MO will be added to the list of managed MOs. During deletion of a MO, the appropriate stable store will be informed of the change, and the MO is removed from the list of managed MOs. As the list of MOs is indexed by OID, MOM needs to be informed of a change to the OID of a MO. The list location is updated upon a change. The mapping management hook is provided for a MO to ask that it be mapped into volatile memory.

In order to manage stable stores, MOM keeps a list of them and allows creation of additional stores only through the MOM public functions. The stable stores are asked for persistent space whenever a new MO is constructed. The stable stores are kept in a list which is saved in the MOM file and recovered when the site comes up. Whenever a new store is created, the list is appended and the MOM file modified.

The role of manager of volatile memory requires that MOM handles volatile replacement. A list of MOs which are currently mapped is kept by MOM. A replacement policy works through this list of MOs, unmapping them to ensure that volatile memory is available when needed.

As the list of MOs managed by MOM is indexed by OID, the associative operation indexing function is provided for free.

5.3 Extents

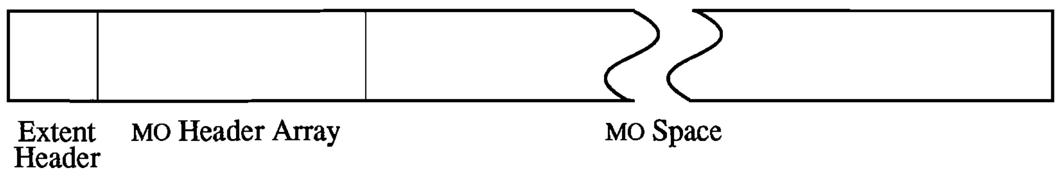

Figure 7: The structure of the extent's stable space.

The management of stable space is the role held by the extent class implementation. The extent allocates stable space for MOs and enacts a replacement policy over this space. As the extent deals with stable space, each extent has a stable file associated with it. This file is broken down as in Figure 7. An array of MO headers is mapped by the extent from the extent file into volatile memory at all times as shown by the stable memory element in Figure 5. Beyond the allocation of a MO header from this array during the construction of a MO, stable space is allocated from the MO space at the end of the extent file. Each MO header is permanently associated with a fixed sized block of stable memory from this MO space. This space can be reallocated to a MO of an equal size following deletion of the MO.

Part of management involves removal of unused resources. Provided the extent is working under a client MOM a stable replacement policy is enacted. The policy works through the array of MO headers, deleting MOs which are unused.

6 Conclusion

MOM provides an interface to storage classes which use the BOSS system. MOM provides a two simple classes in a simple metaphor. The bookkeeping of manually mapping has been hidden from the storage class, and the functionality is cleaner. As an additional bonus, MOM has an indexing function, so callers have no need to remember MOs if they already have the OID.

A Users Guide

The users guide describes the public functions available to the user. Everything related to the MOM is contained in <mom/Mom.h> and <mom/Mo.h>, so inclusion of those files is necessary to use these functions. The users guide is split into three sections, the first contains the standard operations that will be sufficient for most storage classes. The next section touches on a few more functions which are not strictly necessary, though useful. The last section contains unusual functions which are public, but whose invocation will be highly irregular.

A.1 Standard Operations

This section includes all operations that will typically be needed by the average user. The commands include the MOM constructor, MO constructor/destructor, MO pin/unpin, MO loc/size/oid/mark_read.

```
Mom::Mom(int flag,        // 0 for client, 1 for server
         int virt,        // Amount of volatile memory mom can populate
         int stable,      // Amount of stable memory to create if not extant
         int maxmos,      // Maximum number of mos in initial extent
         char *path)      // Name of the initial path
```

The standard operation to fire up a MOM is to create or reload a MOM. The new operation will search if a MOM already exists. If it does, it will ignore all other given arguments and load the existing file. The procedure will also load any stable memory files which are named in the file.

If the file does not exist, the operation will assume that it is to create a new MOM file by the pathname given as the fifth argument. The given arguments are used in this case. The first argument flags whether the new MOM is to be a client or server. The second argument tells the amount of volatile memory the MOM should constrain herself to. The third and fourth arguments are only used in the event that an extent of the given path does not exist. When creating a new stable space, the third argument is its stable size, and the fourth argument is the maximum number of MOs allowed in the stable array.

One warning when creating a MOM. The stable memory should exceed the available volatile memory by the size of the largest block that can be anticipated. If not, stable replacement might not occur as all replaceable MOs could be held by the volatile memory manager and be thus unavailable to the stable memory manager.

```
new (size) Mo::Mo(oid)
Mo::~Mo()
```

Construction and destruction of a Memory Object is fairly straightforward, however the new operation for a MO is overloaded presenting a possibly bizarre syntax. When a new MO is desired the user should use the following format:

```
new ((int) size) Mo((ObjID) oid)
```

This syntax will give the storage class a MO which has a storage space of size size and an ObjID of oid. A stable memory location will first be allocated, and then the MO will be pinned into volatile memory. Due to the operation of the constructor, the user should unpin() or delete the MO as soon as its immediate use is done. If the MO is left pinned, the volatile memory space held by the MO cannot be replaced by MOM. Unpinning the MO will keep the MO in stable storage for an indeterminate amount of time on a client site or until deleted on either type of site.

The delete operation should be used whenever the MO is no longer needed.

```
void *Mo::pin()
void  Mo::unpin()
```

The pin() operation will pin the MO into volatile memory. If the object is not in volatile memory, it will map the MO in from stable storage before pinning it. If it is unable to pin the MO, a pointer to NULL will be returned. Otherwise, a pointer to the volatile memory location will be returned. This memory space can be cast or used any way the user desires provided it does not exceed the size of the MO (obtainable via Mo::size()). The MO will stay in volatile memory until it is unpinned or deleted.

The unpin() operation frees the pin keeping the MO into volatile memory. In addition, it calls an operation to asynchronously synchronize the page in case of a crash. Once the unpin() operation is called, there is no guarantee as to whether the MO is mapped from stable into volatile memory, is only in stable memory, or, in the case of a client site, no longer exists at the site. Unpin() should be called whenever the immediate need for a MO is gone.

```
void *Mo::loc()
int   Mo::size()
ObjID Mo::oid()
void  Mo::mark_read()
```

Calling loc() on a MO will give the volatile memory location of the MOs memory space, if the MO is in volatile memory. As a result of multi-threading, loc() can theoretically return a memory location which will not be valid by the time the storage class accesses it as loc() does not guarantee a pinned MO. In the event that the MO is not in volatile memory, it will return a pointer to NULL.

Size() will return the size of the memory space associated with the MO.

Oid() will return the object identifier associated with the MO in question.

Mark_read() has the effect of telling the MO that the storage class has touched it recently. This operation will lower the probability that the MO will be cleaned out of memory by MOM replacement policies. A MO should be marked as read even if the user has the object pinned, since an unread, pinned MO has a lower replacement priority than read MO s, though it will not be arbitrarily removed from memory.

A.2 Advanced Operations

The advanced operations are not needed by all users, but are useful nonetheless. Commands include the [] indexing function, an assignment operation for oids, a command to query if the MO is pinned and a command to unpin all MOs currently pinned by the MOM.

```
Mo    *Mom::operator [] (ObjID)
```

Every MOM holds a dictionary associating all its MOs with their object identifiers. The user can avail themselves of this dictionary by specifying an oid to the MOM as follows:

```
Mo *mo=mom[oid];
```

This command will check the given oid to see if there is a corresponding Mo. If there is, the MO will be returned. If there is not, a pointer to NULL will be returned. This function was often used as follows:

```
if (!(Mo *mo=mom[oid]))
        // doesn't exist, deal
        ;
else if (!(void *loc=mo->pin()))
        // can't pin, deal
        ;
else {
        // manipulate Mo space at loc as desired
        mo->unpin();
     }
```

These lines will pin the MO associated with the given oid into volatile memory where the memory space can be manipulated however the user desires.

```
ObjID Mo::oid(ObjID)
bool  Mo::is_pinned()
```

The oid(ObjID) function allows the user to assign a new oid to a MO. The new value will be recorded in the MO, the assignment will be persistent and the MOM will be appropriately notified.

Is_pinned() returns a boolean value as to whether the MO in question is pinned into volatile memory. It returns a 1 if it is, and a 0 if not.

```
void   Mom::massunpin()
```

Massunpin() is called on the MOM and is a possibly destructive act. It tells the MOM to unpin all MOs which are currently mapped. The danger lies with multiple threads connected to the same MOM when one thread unpins MOs which might be needed by another thread. This occurrence could result in a segment violation.

A.3 Unusual Operations

```
int    Mom::createExtent(int size,        // size of extent stable memory
                         int maxmos,      // maximum mos allowed
                         char *path)      // pathname of extent
int    Mom::removeExtent(char *path)
```

These two operations will be called by storage classes infrequently at best. CreateExtent() creates an extent by the given pathname. If the pathname already exists, a new extent will not be created, the path there will merely be loaded. In the other case, an extent of the given path with a stable memory size of the given size and a maximum number of MOs equal to maxmos will be created. MOM will know about the new extent until the extent is removed. The eventual size of this new extent will be equal to size+getpagesize()+maxmos*sizeof(MO). A negative return value signifies an error.

RemoveExtent() will remove all MOM knowledge of the specified pathnamed extent. If the file does not exist to MOMs knowledge, a 0 will be returned. The extent file is not removed from the filesystem. RemoveExtent() should almost never be called as the user cannot know which extents hold which MOsand MOs would be mistakenly deleted from MOM knowledge.

```
void   Mom::~Mom()
```

The MOM destructor is the safe way to shut it down. All MOs are unpinned, everything is unmapped, extents are closed and the MOM shuts down. When the MOM is brought back up again via new, all the extents and MOs will be reloaded.

B Acknowledgments

The author would like to thank David E. Langworthy for the gracious permission to use some of his many graphic elements in the production of this paper, and for his direction and management throughout this project. It would have been at least poorly written, if not impossible without him.

References

[1] D. Langworthy. Memory Object Management in the Brown Object Storage System. Unpublished paper (as of yet), Department of Computer Science, Brown University, 1994.

```
// This may look like C code, but it is really -*- C++ -*-

#ifndef MO_H
#define MO_H

/*************************************************************************/
// Memory Object Sub (Mosub) Header
// February 10, 1994
//
// Vincent C. Rubino
//
// The point of a Mo is to provide persistent memory to the user that can
// be moved in and out of volatile memory quickly.
/*************************************************************************/

/* Include BOSS Headers */
#include <bo/BOSS.h>
#include <mom/Mom.h>

/*************************************************************************/
/*                                                                       */
/*      Class Mo                                                         */
/*                                                                       */
/*************************************************************************/

class Mo
{
friend class Mom;
friend class Extent;

  public:

    void *operator new(size_t,          // get memory from Mom for class,
                       int);            // cleaning Mo\'s if insuff space
                                        // also get block of size int

    void  operator delete(void *p);     // free memory for
                                        // class.

    Mo(ObjID);                  // get a virtual and stable pinned block
                                // of size int from Mom;

    ~Mo();                      // release the stable block and clean
                                // up extraneous;

    enum { read_v = 0, unread_v, read_s, unread_s};
    enum { pinned=0, unpinned, doomed};

    void  *loc()                { if (_pinned==doomed) return NULL;
                                  else return _virtuallocation; }
    void  *pin();               // pin the Mo
    void   unpin();             // unpin the Mo

    ObjID  oid()                { return _oid; }
    ObjID  oid(ObjID oid);      // changed the oid of the Mo

    void   mark_read();         // mark the Mo as having been recently read
    int    size()               { return _size; }

    static void set_mom(Mom *m) { _mom = m;}

    int    virtsize();          // returns the virtual memory
                                // taken by mapping the mo.
    bool    is_pinned()          { return (_pinned==pinned); }

  private:
    void    doom()               { _pinned=doomed; }

    void    set_size(int s)      { _size=s; }
    void    set_stable(off_t l)  { _stablelocation=l; }
    void    set_loc(void *newloc){ _virtuallocation=newloc; }

    off_t   stableloc()          { return _stablelocation; }

    Extent *extent()             { return _extent; }
    void    extent(Extent *ext)  { _extent=ext; }

    void    mark_pinned()        { _pinned=pinned; }
    void    mark_unpinned()      { _pinned=unpinned; }
    void    force_reads()        { _state=read_s; }
    void    force_unread()       { force_unreadv();
                                   force_unreads(); }
    void    force_unreadv()      { if (is_readv())
                                       _state=unread_v; }
    void    force_unreads()      { if (is_reads())
                                       _state=unread_s; }

    void   *unsafePin();         // calls made if the lock is known to be held
    void    unsafeUnpin();       // calls made if the lock is known to be held

    void    mark_unread();       // mark as being unread
    void    mark_unreads();      // mark as being unreads
    void    mark_unreadv();      // mark as being unreadv

    bool    is_read()            { return ((_state==read_v) ||
                                           (_state==read_s)); }
    bool    is_reads()           { return (_state==read_s); }
    bool    is_readv()           { return (_state==read_v); }
    bool    is_unreadv()         { return (_state==unread_v); }
    bool    is_unreads()         { return (_state==unread_s); }
    bool    is_doomed()          { return (_pinned==doomed); }
    bool    is_unpinned()        { return (_pinned==unpinned); }

  protected:                     //  This space is obtained from _mom
    static Mom  *_mom;

    ObjID        _oid;
    off_t        _stablelocation;
    Extent      *_extent;
    void        *_virtuallocation;
    int          _size;
    unsigned int _state  : 3;
    unsigned int _pinned : 2;
    unsigned int _rank   : 27;

    };

#endif /* MO_H */
```

```
// This may look like C code, but it is really -*- C++ -*-

#ifndef _MOM_H
#define _MOM_H

/**************************************************************************/
// Memory Object Manager (MOM) Header
// February 10, 1994
//
// Vincent C. Rubino
//
// The point of the MOM is to manage the flow of stable to volatile memory
// via the extent and MO constructs.  It takes care of volatile replacement.
/**************************************************************************/


#include <LEDA/dictionary.h>
#include <fcntl.h>
#include <sys/mman.h>

/* Include BOSS Headers */
#include <bo/BOSS.h>
#include <nm/ObjID.h>
#include <mtcp/MTCP_Thread.H>

extern "C" {
    int getpagesize();
};

extern int compare(const ObjID&, const ObjID&);

/**************************************************************************/
/*                                                                        */
/*     Class Mom                                                          */
/*                                                                        */
/**************************************************************************/

const int _MAX_CACHED_MOS  = 2*1024;

class Mom {
friend class Mo;
friend class Extent;

  public:
    enum { client = 0, server };
    static int _PAGE_SZ;

    Mom(int flag, int virt,              // open/create Mom file
        int stable, int maxmos,
        char *name);

    ~Mom();                              // clean up the destruction of the Mom

    int   createExtent(int=100*1024*1024, // Create extent of specified path
                       int maxmos=1024*1024,
                       char* ="_testExt");

    int   removeExtent(char* ="_testExt");// Close extent from Mom

    Mo   *operator [] (ObjID);           // Return the meMory object for
                                         // the oid

    int   isserver()                     { return (_isserver==TRUE); }
    static void lock()                { _mom_lock.acquire(); }
    static void unlock()              { _mom_lock.release(); }

    void   massunpin();               // unpin every mo associated
                                      // with the thread

  private:
    void *initMo(size_t,              // get stable space for Mo information
                 int=4*1024);         // also get a block of size int

    void  delMo(void *);              // delete stable space for Mo
                                      // information after clearing it
                                      // from volatile memory

    void  changeObjId(ObjID, ObjID);  // set first ObjID Mo to second value
    void  registerObjId(ObjID, Mo *); // register ObjID for the given Mo

    void *mapBlock(Mo *);             // Notifies Mom that Mo needs to
                                      // be in volatile memory.
                                      // Mom will clear up volatile
                                      // space, mmap the block and return
                                      // pointer to block

    void  unmapBlock(Mo *);           // get Mo * out of virtual memory
    void  unmapBlock(Mo *, int);      // unmap mo from virt mem from given
                                      // place in pinnedarray

    int   addPinnedList(Mo *);        // open space in pinnned list, then
                                      // add mo to it.

    void  dumpMomfile();              // will dump the mom file to disk
                                      // will trash any file that is there
    void  loadMomfile();              // loads the mom file from disk
                                      // trashing all info in volatile
                                      // memory

    off_t alignOffset(off_t);         // gives the page offset before off_t
    int   alignSizeModifier(off_t);   // gives the distance of off_t to
                                      // beginning of its page

    void  checkForCleaning(int);      // clean given mo if it can be cleaned
    void  prepareForCleaning(int);    // mark mo as unread

    void  repopulateMolist();         // Hack to get around thread package

  protected:                          // Some are persistent,
                                      // most are calculated

    // Persistent information
    int   _vsize;                     // size of total Mom virtual memory
    int   _numextents;                // number of extents
    int   _isserver;                  // flag if it is a server or a client

    // Volatile information
    char  _Mompath[100];              // path of the Mom file
    int   _virtfree;                  // size of allocable Mom virtual memory
    int   _currentextent;             // current extent for walking
    int   _num_mapped_mos;            // number of mapped mos

    static Thread_Lock _extlock;      // lock for extent dictionary
    static Thread_Lock _molistlock;   // lock for molist dictionary

    dictionary<int, Extent*> _extents; // head of the list of extents
                                      // iterate* and search for name
    dictionary<ObjID, Mo *> _molist;  // list of managed Mo\'s
```

```
                                        // search on ObjID

    Mo    *_pinnedarray[_MAX_CACHED_MOS];// head of list of cached Mo\'s


    static Thread_Lock _mom_lock;       // lock for mom status stuff

    int   _currentmo;                   // current mo in the pinned array
    };

#endif /*_MOM_H */
```

```
// This may look like C code, but it is really -*- C++ -*-

#ifndef _EXTENT_H
#define _EXTENT_H

/*************************************************************************/
// Extent Header
// March 15, 1994
//
// Vincent C. Rubino
//
// The point of the Extent object is to keep track of the stable memory space
// for the mom.  Basically a mom will have at least one extent and will
// be the only location where extents can be added.  Extents can
// be loaded, created and allocated from.  The extent package takes
// care of cleaning out the stable memory.
//
// Quick overview on the file structure of an extent:
// |_| |__________| |___________________________ ... ____________________|
// EI   MoIndex     Memspc
//
// EI contains persistent extent information
// MoIndex is the space allocated for internal Mo information
// Memspc is the memory space given given to Mo\'s.
//
// For each MoIndex entry there is one memory space.  memspc is a multiple
// of 8 bytes.
/*************************************************************************/

/* Include BOSS Headers */
#include <bo/BOSS.h>
#include <sys/types.h>
#include <mom/Mo.h>
#include <mom/Mom.h>

/*************************************************************************/
/*                                                                       */
/*      Class Extent                                                     */
/*                                                                       */
/*************************************************************************/

class Mo;

static char *_testExtent="/tmp/test.ext";

class Extent
{
friend Mom;

  public:
    Extent(char* =_testExtent,            // path for the extent
           int=50*1024*2,                 // maximum number of mo\'s
           int=50*1024*1024);             // maximum total space for extent
    ~Extent();

    void *allocMo(int=512);               // try to allocate space for Mo
                                          // and memspc of size int
                                          // return Mo or NULL--NULL chk errno
                                          // -1 err; 0 can\'t; 1 can;
                                          // -2 over/under totblk ie next Ext

    bool  is_path(char *path)             { return (!strcmp(path, _path)); }
    char *path()                          { return _path; }
    int   size()                          { return *_size; }
    int   vsize();                        // return virtual space taken by ext
    int   fd()                            { return _fd; }

    static void set_mom(Mom *m)           {_mom=m;}
    static void set_page(int pg)          {_PAGE_SZ=pg;}

    void  sync();                         // asychronously sync the _molist
    void  sync(Mo *);                     // synch the specified Mo

  private:
    int cleanMo(int monum);               // mark mo as cleaned
                                          // -1 err; 0 can\'t; 1 can;
    int canDoAllocMo(int monum, int sz);// check if mo can be allocated
                                          // -1 err; 0 can\'t; 1 can

    void assignExtentVars();              // set all the int *\'s to the
                                          // correct locations
    void initializeExtentVars(char *,     // put initial values into extent
                              int, int);
    int  mapExtentVars();                 // map extent information
    int  mapMolist();                     // map molist
    void fillMomMolist();                 // fill mom with mo\'s
    void unsafeFillMomMolist();           // file mom with mo\'s once locked
    void unsafeRecoverMos();              // called as part of recovery
    int  createExtentFile(char *, int); // make a new extent
    void *allocInitialMo(int);            // Allocate a mo from free space
    void dumpMos();                       // dump extent information
                                          // for debugging

  protected:
    static Mom *_mom;                     // pointer to mom
    static int _PAGE_SZ;                  // page size for the process

    int         _fd;                      // file descriptor of Extent
    int        *_size;                    // total size of extent  (xxx save)
    off_t      *_memspcoffset;            // offset of start of memory space
    int        *_maxnummos;               // maximum number of mo\'s in extent
    int        *_nummos;                  // current number of mo\'s in extent
    off_t      *_freememspcoffset;        // starting pointer of free memspace
    int         _currentmonum;            // current mo for allocation

    char       *_path;                    // the name of the path

    void       *_mappedindex;             // total mapped index area

    Mo         *_molist;                  // Mo\'s w/in the extent
};

#endif  /* _EXTENT_H */
```

```
// This may look like C code, but it is really -*- C++ -*-

/************************************************************************/
// MeMory Object (Mo) Source
// February 10, 1994
//
// Vincent C. Rubino
//
// The point of a Mo is to provide persistent meMory to the user that can
// be Moved in and out of volatile meMory quickly.
/************************************************************************/

#include <mom/Mo.h>
#include <mom/Extent.h>
#include <signal.h>

#ifdef Mo_DEBUG

Thread_Lock PRINT_LOCK;
FILE *DEBUG_file = 0;

#endif

/************************************************************************/
/*                                                                      */
/*      Class Mo                                                        */
/*                                                                      */
/*  Syntax for creating:  new (size) Mo(objid);                         */
/************************************************************************/

void *Mo::operator new(size_t mosize, int size)
{
    void *temp=_mom->initMo(mosize, size);
    if (temp==NULL)
        cerr << "Could not Allocate!" << endl;
    return (temp);
}

void  Mo::operator delete(void *mo)
{
    _mom->delMo(mo);
}

Mo::Mo(ObjID objid)
{
    _oid=objid;
    _mom->registerObjId(objid, this);
    _pinned=unpinned;
    _virtuallocation=NULL;
    pin();
}

int  Mo::virtsize()
{
    return _size;
}

Mo::~Mo()
{
}

void *Mo::unsafePin()
{
    if ((_pinned==pinned) && (_virtuallocation)) {
        _state=read_v;
        return _virtuallocation;
    }
    if (_pinned==doomed) {
        // Cant do anything with a doomed object.
        cerr << "Attempt to pin doomed object" << endl;
        return NULL;
    }
    if (_virtuallocation==NULL) {
        if ((_virtuallocation=_mom->mapBlock(this)) == NULL) {
            cerr << "mapBlock failed on " << _oid << endl;
            return NULL;
        }
    }
    _pinned=pinned;
    _state=read_v;
    return _virtuallocation;
}

void *Mo::pin()
{
    if ((_pinned==pinned) && (_virtuallocation)) {
        _state=read_v;
        return _virtuallocation;
    }
    _mom->lock();
    if (_pinned==doomed) {
        // Cant do anything with a doomed object.
        cerr << "Attempt to pin doomed object" << endl;
        _mom->unlock();
        return NULL;
    }
    if (_virtuallocation==NULL) {
        if ((_virtuallocation=_mom->mapBlock(this)) == NULL) {
            cerr << "mapBlock failed on " << _oid << endl;
            _mom->unlock();
            return NULL;
        }
    }
    _pinned=pinned;
    _state=read_v;
    _mom->unlock();
    return _virtuallocation;
}

void  Mo::unpin()
{
    if (_pinned!=pinned)
        return;
    _mom->lock();
    if (_pinned!=doomed)
        _pinned=unpinned;
    _mom->unlock();
    _extent->sync();
}

void  Mo::unsafeUnpin()
{
    if (_pinned!=doomed)
        _pinned=unpinned;
    _extent->sync();
}

void  Mo:: mark_read()
```

```
{
    if (_state==read_v)
        return;
    _mom->lock();
    if (_state==unread_v)
        _state = read_v;
    else if (_state==unread_s)
        _state = read_s;
    _mom->unlock();
}

void  Mo::mark_unread()
{
    if ((_state==unread_v) || (_state==unread_s))
        return;
    _mom->lock();
    force_unread();
    _mom->unlock();
}

void  Mo::mark_unreads()
{
    if (_state!=read_s)
        return;
    _mom->lock();
    force_unreads();
    _mom->unlock();
}

void  Mo::mark_unreadv()
{
    if (_state!=read_v)
        return;
    _mom->lock();
    force_unreadv();
    _mom->unlock();
}

ObjID  Mo::oid(ObjID oid)
{
    _mom->lock();
    _mom->changeObjId(_oid, oid);
    _oid=oid;
    _mom->unlock();
    return _oid;
}
```

```
// This may look like C code, but it is really -*- C++ -*-

/*********************************************************************
 * Memory Object Manager (MOM)                                       *
 * February 10, 1994                                                 *
 *                                                                   *
 * Vincent C. Rubino                                                 *
 *                                                                   *
 *********************************************************************/

#include <mom/Mom.h>
#include <class/StorageClass.h>
#include <signal.h>
#include <mom/Mo.h>
#include <mom/Extent.h>
#include <nm/ClsID.h>

#ifdef MOM_DEBUG

Thread_Lock PRINT_LOCK;
FILE *DEBUG_file = 0;

#endif

#define MOM_MAX_MOS_STABLE 2000
Mo *_moarray[MOM_MAX_MOS_STABLE];       // vcr hack replacing molist

/*********************************************************************/
/*                                                                   */
/*      Class Mom                                                    */
/*                                                                   */
/*********************************************************************/
Thread_Lock Mom::_mom_lock;
Thread_Lock Mom::_extlock;
Thread_Lock Mom::_molistlock;

#ifndef SOLARIS
int Mom::_PAGE_SZ = getpagesize();
#else
#include <unistd.h>
int Mom::_PAGE_SZ = sysconf(_SC_PAGESIZE);
#endif

Mom::Mom(int flag, int virtualmemsize, int maxmos, int stablememsize,
         char *path)
:_currentmo(0)
{
    cerr << "Calling Mo::set_mom and Extent::set_mom setting them to: "
         << (u_long) this << endl;
    Mo::set_mom(this);
    Extent::set_mom(this);
    Extent::set_page(_PAGE_SZ);

    for (int i=0; i<_MAX_CACHED_MOS; i++)
        _pinnedarray[i]=NULL;

    // open Mom file

    sprintf(_Mompath, "%s.mom", path);
    int test=open(_Mompath, O_RDWR, 666);
    if (test<0) {       // file probably doesn't exist
        cerr << "Mom cannot be opened, creating ";
        if (flag==client) cerr << "client" << endl;
        else cerr << "server" << endl;
        // Initialize internal Mom information

        _currentextent=_numextents=0;
        _virtfree=_vsize=virtualmemsize;
        if (flag==client)
            _isserver=FALSE;
        else
            _isserver=TRUE;

        // save information to the file
        dumpMomfile();

        // check for error
        if (createExtent(maxmos, stablememsize, path)<0) {
            perror("Fatal:  Cannot create extent in new Mom");
            kill(getpid(), SIGSTOP);
            return;
        }

        cerr << "Mom file and initial extent have been created" << endl;
    }   // end if file doesn\'t exist
    else {
        cerr << _Mompath << " Mom file exists.  Loading mom" << endl;
        close(test);
        loadMomfile();
        _currentextent=0;
    }   // end if file does exist

}

Mom::~Mom()
{
    // unpin all the mo\'s
    massunpin();

    // Just delete all the extents
    lock();
    for (int i=0; i<_numextents; i++) {
        Extent *ext=_extents.inf(_extents.lookup(i));
        if (ext!=NULL)
            delete ext;
    }
    unlock();
}

void  Mom::massunpin()
{
    lock();
    for (int i=0; i<_MAX_CACHED_MOS; i++)
        if (_pinnedarray[i]!=NULL) {
            Mo *mo=_pinnedarray[i];
            mo->unsafeUnpin();
        }
    unlock();
}

int Mom::createExtent(int size, int maxmos, char *path)
{
    char buff[100];
    sprintf(buff, "%s.extent", path);
    cerr << "Creating new extent" << endl;
    Extent *ext=new Extent(buff, maxmos, size);
    if (ext==NULL)
```

```
        return -1;

    //  add Extent to list of Momextents
    _extlock.acquire();
    _extents.insert(_numextents++, ext);

    dumpMomfile();

    _extlock.release();

    cerr << "extent created" << endl;
    return 1;
}

void  Mom::dumpMomfile()
{
    cerr << "Dumping Mom file" << endl;

    // shadow mom file
    char tmpbuf[100];
    sprintf(tmpbuf, "%s.tmp", _Mompath);

    FILE *momfd=fopen(tmpbuf, "w");
    if (momfd==NULL)
        // deal with error
        return;

    fprintf(momfd, "%d\t%d\t%d\n", _isserver, _vsize, _numextents);
    for (int i=0; i<_numextents; i++) {
        dic_item it=_extents.lookup(i);
        if (it) {
            Extent *ext=_extents.inf(it);
            if (ext)
                fprintf(momfd, "%s\n", ext->path());
            else
                fprintf(momfd, "%s\n", "NULL_EXTENT");
        }
        else
            fprintf(momfd, "%s\n", "NULL_EXTENT");
    }
    fclose(momfd);
    // unshadow momfile
    if (rename(tmpbuf, _Mompath)<0)
        perror("Problem in dumpMomfile()");
    cerr << "Mom file dumped" << endl;
}

void  Mom::loadMomfile()
{
    char buf[100];
    cerr << "Loading Mom file" << endl;

    // See if a previous shadow of mom file exists
    sprintf(buf, "%s.tmp", _Mompath);
    FILE *momfd=fopen(buf, "r");
    if (momfd!=NULL) {
        cerr << "Mom file probably out of date" << endl;
        fclose(momfd);
    }

    // open MOM file
    momfd=fopen(_Mompath, "r+");
    if (momfd==NULL) {
        cerr << "Cannot load mom file" << endl;
        kill(getpid(), SIGSTOP);
        // deal with error
        return;
    }

    // load information from MOMfile
    fscanf(momfd, "%d%d%d\n", &_isserver, &_vsize, &_numextents);

    cerr << "Loaded Mom: ";
    if (_isserver) cerr << "server, ";
    else cerr << "client, ";
    cerr << "virtmemsize: " << _vsize << " and " << _numextents
         << " extents" << endl;

    // load extent names, and load extents
    _virtfree=_vsize;
    for (int i=0; i<_numextents; i++) {
        fscanf(momfd, "%s\n", buf);
        cerr << "Loading " << buf << endl;
        if ((strcmp(buf, "NULL_EXTENT"))!=0) {
            Extent *ext=new Extent((char *) buf);
            _extents.insert(i, ext);
            //_virtfree-=ext->vsize();
        }
    }
    fclose(momfd);
    cerr << "Mom file loaded" << endl;
}


int Mom::removeExtent(char *path)
{
    int i;
    int found=FALSE;
    Extent *ext;

    // check if path exists and in Mom extents
    // remove from extent list
    _extlock.acquire();
    for (i=0; i<_numextents; i++) {
        ext=_extents.inf(_extents.lookup(i));
        if (ext!=NULL)
            if (ext->is_path(path)) {
                found=TRUE;
                break;
            }
    }
    if (!found) {
        _extlock.release();
        return 0;
    }
    _extents.del(i);

    // save change to MOMfile
    dumpMomfile();
    _extlock.release();

    // close extent
    delete ext;
    return 1;
}

Mo *Mom::operator[](ObjID objid)
{
    // fixes multi-threading problem.

    static firsttime=1;
    static counter=0;
    if (firsttime) {
        lock();
        if (counter<3) {
            repopulateMolist();
            _currentmo=0;
            firsttime=0;
            counter++;
        }
        unlock();
    }
    return (_moarray[(objid.SegINT())]);
}

void  Mom::repopulateMolist()
{
    // fixed multi-thread problem

    for (int i=0; i<MOM_MAX_MOS_STABLE; i++)
        _moarray[i]=NULL;
    for (i=0; i<_numextents; i++) {
        Extent *ext;
        dic_item it=_extents.lookup(i);
        if (it) {
            ext=_extents.inf(it);
            ext->mapMolist();
            // ext->unsafeRecoverMos();
            ext->unsafeFillMomMolist();
        }
    }
}

void *Mom::initMo(size_t, int blocksize)
{
    int startext=_currentextent-1;
    Extent *ext=NULL;
    void *mo=NULL;
    dic_item it;
    int iterations=0;

    if (startext<0) startext=_numextents+2;

    // Allocate stable space for Mo from extents
    // go through all extents at least twice before giving up.
    _extlock.acquire();
    while (iterations<2) {
        while ((_currentextent!=startext) && (mo==NULL)){
            //cerr << "Looking at extent " << ext << endl;
            if (ext==NULL) {
                if (it=_extents.lookup(_currentextent))
                    ext=_extents.inf(it);
                else {
                    _currentextent++;
                    if (_currentextent>(_numextents+2)) _currentextent=0;
                }
            }
            if (ext!=NULL) {
                void *mo=ext->allocMo(blocksize);
                if (mo==NULL) {
                    _currentextent++;
                    if (_currentextent>(_numextents+2)) _currentextent=0;
                    ext=NULL;
                } else {
                    _extlock.release();
                    //  cerr << "Extunlock" << endl;
                    return mo;
                }
            }
        }
        _currentextent++;
        iterations++;
    }
    _extlock.release();
    kill(getpid(), SIGSTOP);
    return NULL;
}

void  Mom::delMo(void *loc)
{
    // view loc as a Mo *

    Mo *mo=(Mo *) loc;

    // unpin/doom the mo

    mo->doom();

    // delete Mo from _molist

    _molistlock.acquire();
    _moarray[(mo->oid()).SegINT()]=NULL;
    _molistlock.release();

    // if object is in virtual memory
    if ((mo->loc()!=NULL) || (mo->is_unreadv()) || (mo->is_readv())) {
        unmapBlock(mo);
    }

    // smtoss it
    mo->oid().ClsID().getInstance()->smtoss(*mo);

    // set size to -size to signify that it has been deleted
    int tmpsz=-(mo->size());
    mo->set_size(tmpsz);
}


void *Mom::mapBlock(Mo *mo)
{
    int mospc=addPinnedList(mo);
    if (mospc<0) {
        cerr << "Cannot allocate space for mo in virtual memory" << endl;
        return NULL;
    }

    // mmap mo into virtual memory
    Extent *ext=mo->extent();
    void *location=mmap(NULL,
                        (mo->size()+
                             alignSizeModifier(mo->stableloc())),
                        PROT_READ|PROT_WRITE,
                        MAP_SHARED, ext->fd(),
                        alignOffset(mo->stableloc()));
    if (!location) {
        cerr << "Fatal Error: Cannot map Mo" << endl;
```

```
            return NULL;
        }

        // take away total virtual memory
        _virtfree-=mo->virtsize();

        // return the location
        return (void *)((int)location+alignSizeModifier(mo->stableloc()));
    }

    void  Mom::unmapBlock(Mo *mo)
    {
        // find mo in pinned list
        for (int i=0; i<_MAX_CACHED_MOS; i++)
            if (_pinnedarray[i]==mo)
                unmapBlock(mo, i);
    }

    void  Mom::unmapBlock(Mo *mo, int cachedloc)
    {
        // pull mo from pinnedarray[cachedloc]
        _pinnedarray[cachedloc]=NULL;

        // increment size of available virtual memory
        _virtfree+=mo->virtsize();

        // vmtoss it
        mo->oid().ClsID().getInstance()->vmtoss(*mo);

        // munmap mo from virtual memory
        munmap((caddr_t) ((int)(mo->loc())-alignSizeModifier(mo->stableloc())),
               mo->size()+alignSizeModifier(mo->stableloc()));

        // set mo->_virtuallocation to NULL
        mo->set_loc(NULL);

        // set mo to read_s and unpinned.
        mo->force_reads();
        mo->mark_unpinned();
    }

    int   Mom::addPinnedList(Mo *mo)
    {
        int temp=_currentmo-1;
        if (temp<0) temp=_MAX_CACHED_MOS-1;
        while ((_currentmo!=temp)/* || (isserver()!=TRUE)*/) {
            // if extant, mark 50% away from currentmo for cleaning
            prepareForCleaning(_currentmo);

            if (_pinnedarray[_currentmo]!=NULL)     // clean if possible
                checkForCleaning(_currentmo);

            // if (_currentmo==NULL && enough volatile space)
            // assign and return
            if ((_pinnedarray[_currentmo]==NULL) &&
                (_virtfree>=mo->virtsize())) {
                _pinnedarray[_currentmo]=mo;
                return 0;
            }
            // increment currentmo;
            if ((++_currentmo)>=_MAX_CACHED_MOS)
                _currentmo=0;
        }   // end endless
        // virtual memory is full
        return -1;
    }

    void  Mom::checkForCleaning(int mo)
    {
        // if unreadv and unpinned, doom it
        if (_pinnedarray[mo]==NULL)
            return;
        if ((_pinnedarray[mo]->is_unreadv()) &&
            (_pinnedarray[mo]->is_unpinned())) {

            _pinnedarray[mo]->doom();

            // unmapBlock
            unmapBlock(_pinnedarray[mo], mo);
        }   // end if statement
    }

    void  Mom::prepareForCleaning(int mo)
    {
        int temp=(mo+(_MAX_CACHED_MOS>>1))&(_MAX_CACHED_MOS-1);

        if (_pinnedarray[temp]!=NULL)
            if (_pinnedarray[temp]->is_readv()) {
                _pinnedarray[temp]->force_unreadv();
            }
    }

    off_t Mom::alignOffset(off_t loc)
    {
        // align the given offset to the start of the appropriate page
        return ((loc>>12)<<12);
    }

    int   Mom::alignSizeModifier(off_t loc)
    {
        // return size modifier for the given offset
        // ie  loc==200 aligns to 0, so size modifier is 200.
        return (loc&4095);
    }

    void  Mom::changeObjId(ObjID old, ObjID changed)
    {
        _molistlock.acquire();
        _moarray[changed.SegINT()]=_moarray[old.SegINT()];
        _moarray[old.SegINT()]=NULL;
        _molistlock.release();
    }

    void  Mom::registerObjId(ObjID oid, Mo *mo)
    {
        _molistlock.acquire();
        _moarray[oid.SegINT()]=mo;
        _molistlock.release();
    }
```

```
// This may look like C code, but it is really -*- C++ -*-

/*************************************************************************/
// Extent Source
// February 10, 1994
//
// Vincent C. Rubino
//
// The point of an Extent is to provide stable meMory to the mom that can
// be allocated easily and cleaned up without her knowing
/*************************************************************************/

#include <mom/Extent.h>
#include <fixed/Fixed.h>
#include <signal.h>
#include <errno.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <syscall.h>

extern "C" {
int syscall(int code,...);
}

int Mom::_PAGE_SZ;

#ifdef Mo_DEBUG

Thread_Lock PRINT_LOCK;
FILE *DEBUG_file = 0;

#endif

#define MAP_FAILED -1

/*************************************************************************/
/*                                                                       */
/*      Class Extent                                                     */
/*                                                                       */
/*************************************************************************/

Extent::Extent(char *path, int maxmos, int size)
{
    // open path

    cerr << "opening an extent.  Args:" << path << ", " << maxmos << ", "
         << size << endl;
    int initial=_fd=syscall(SYS_open, path, O_RDWR, 0666);
    if (_fd<0) {        // file probably doesn\'t exist
        cerr << "extent does not exist, will create" << endl;
        // create file
        if (createExtentFile(path, (size+_PAGE_SZ+maxmos*sizeof(Mo)))<0)
            kill(getpid(), SIGSTOP);
    }

    // mmap _freeblocks, _firstfreeblock, _maxblocks and _size
    // save _size _freeblocks by mapping...

    if (mapExtentVars()<0)
        kill(getpid(), SIGSTOP);

    // make pages a little more accessible

    assignExtentVars();
    if (initial<0) {
        // initialize size, firstfreeblock, freeblocks and maxblocks
        // init _size, memspcoffset, maxnummos, nummos, freememspcoffset

        initializeExtentVars(path, maxmos,
                             (size+_PAGE_SZ+maxmos*sizeof(Mo)));
    }

    _molist=NULL;

    if (mapMolist()<0)
        kill(getpid(), SIGSTOP);

    if (initial>=0) {
        // fill up mom with all the current Mo\'s
        _mom->lock();
        unsafeRecoverMos();
        fillMomMolist();
        _mom->unlock();
    }
    else
        cerr << "$ initial == 0\n";
    // initialize information

    _currentmonum=0;

}

Extent::~Extent()
{
    // ~Extent is used to close the Extent file.

    // munmap mappedindex and list of mo information
    if (munmap((caddr_t) _molist, *_maxnummos*sizeof(Mo))<0)
        // deal with error;
        return;
    if (munmap((caddr_t) _mappedindex, _PAGE_SZ)<0)
        // deal with error
        return;
    // close Extent file
    close(_fd);
}

int  Extent::createExtentFile(char *path, int size)
{
    _fd=syscall(SYS_open, path, O_CREAT | O_RDWR, 0666);
    if (_fd<0) {        /*              // major error */
        perror("Cannot create extent file");
        return -1;
    }
    if (ftruncate(_fd, size)<0) {
        perror("Cannot truncate file");
        return -1;
    }
    return 1;
}

void  Extent::initializeExtentVars(char *path, int maxmos, int size)
{
    *_size=size;
    *_nummos=0;
    *_maxnummos=maxmos;
    strcpy(_path, path);
    *_freememspcoffset=*_memspcoffset=
        (_PAGE_SZ+*_maxnummos*sizeof(Mo));
}

void  Extent::assignExtentVars()
{
    _size=(int *)_mappedindex;
    _memspcoffset=(off_t *)((int)_size+sizeof(int));
    _maxnummos=(int *)((int)_memspcoffset+sizeof(off_t));
    _nummos=(int *)((int)_maxnummos+sizeof(int));
    _freememspcoffset=(off_t *)(_nummos+sizeof(int));

    _path=(char *) (_freememspcoffset+sizeof(off_t));
}

int  Extent::mapMolist()
{
    if (_molist!=NULL) {
        if ((_mappedindex = (int *) mmap((caddr_t) _mappedindex,
                                         _PAGE_SZ,
                                         PROT_READ|PROT_WRITE,
                                         MAP_SHARED|MAP_FIXED,
                                         _fd,
                                         0)) == (int *) MAP_FAILED ) {
            perror("Cannot remap Extent header");
            return -1;
        }
        assignExtentVars();
        munmap((caddr_t) _molist, *_maxnummos*sizeof(Mo));
        Mo *tmp= (Mo *) mmap((caddr_t) _molist,
                             *_maxnummos*sizeof(Mo),
                             PROT_READ|PROT_WRITE,
                             MAP_SHARED|MAP_FIXED,
                             _fd,
                             _PAGE_SZ);
        if (tmp==(Mo *) MAP_FAILED)  {
            perror("Extent::mapMolist() failed:  unable to mmap");
            return -1;
        }
        else {
            _molist=tmp;
        }
    }
    else {
        _molist=(Mo *) mmap((caddr_t) _molist,
                            *_maxnummos*sizeof(Mo),
                            PROT_READ|PROT_WRITE,
                            MAP_SHARED,
                            _fd,
                            _PAGE_SZ);
    }
    cerr << "Mo list stable at " << _PAGE_SZ << " and virtual at "
         << (void *) _molist << endl;
    return 0;
}

int   Extent::mapExtentVars()
{
    if ((_mappedindex = (int *) mmap(NULL,
                                     _PAGE_SZ,
                                     PROT_READ|PROT_WRITE,
                                     MAP_SHARED,
                                     _fd,
                                        0)) == (int *) MAP_FAILED ) {
            perror("Cannot mmap within extent constructor");
            cerr << errno << endl;
            return -1;
        }
        return 0;
    }

    void  Extent::fillMomMolist()
    {
        // fill up mom with all the current Mo\'s
        _mom->_molistlock.acquire();
        unsafeFillMomMolist();
        _mom->_molistlock.release();
    }

    void  Extent::unsafeFillMomMolist()
    {
        int i;
        Mo *mo;

        cerr << "Filling MomMolist" << endl;
        for (i=0; i<*_nummos; i++) {
            mo= _molist + i;
            if (mo->size()>0) {
                // inserts the mo in the momlist as a side effect
                _mom->registerObjId(mo->oid(), mo);
            }
            else
                cerr << "Neg Sz: $ " << mo->oid() << endl;
        }
    }

    void  Extent::unsafeRecoverMos()
    {
        int i;
        Mo *mo;
        cerr << "RecoveringMos" << endl;
        for (i=0; i<*_nummos; i++) {
            mo= _molist + i;
            if (mo->size()>0) {
                mo->set_loc(NULL);
                mo->_extent=this;
                // vcr what if it died while doomed?
                mo->mark_unpinned();
                mo->force_reads();
            }
            else
                cerr << "Neg Sz: $ " << mo->oid() << endl;
        }
    }

    void  Extent::sync()
    {
        msync((caddr_t) _molist, *_maxnummos*sizeof(Mo), MS_ASYNC);
    }

    void  Extent::sync(Mo *mo)
    {
        // implemented, but not necessary.
    }

    void *Extent::allocInitialMo(int size)
    {
```

```
    _mom->lock();
    cleanMo(*_nummos);

    Mo *mo=_molist + *_nummos;
    mo->set_size(size);
    mo->_extent=this;
    mo->_stablelocation=*_freememspcoffset;
    *_nummos+=1;
    *_freememspcoffset+=size;
    _mom->unlock();

    return (void *) mo;
}

void  Extent::dumpMos()
{
    cerr << "Dumping all " << *_nummos
         << " Mos in Extent " << this << endl;
    for (int i=0; i<*_nummos; i++) {
        Mo *tmpmo=_molist+i;
        if (tmpmo) {
            cerr << tmpmo->oid();
            cerr << "Size:  " << tmpmo->size();
            if (tmpmo->is_pinned())
                cerr << ", pinned";
            if (tmpmo->is_unpinned())
                cerr << ", unpinned";
            if (tmpmo->is_doomed())
                cerr << ", doomed";
            if (tmpmo->is_readv())
                cerr << ", readv";
            if (tmpmo->is_unreadv())
                cerr << ", unreadv";
            if (tmpmo->is_reads())
                cerr << ", reads";
            if (tmpmo->is_unreads())
                cerr << ", unreads";
            cerr << endl;
        }
    }
    cerr << "Mo's dumped" << endl;
}

void *Extent::allocMo(int size)
{
    // deal with initial allocation
    if (size%8!=0) {
        cerr << "Not size of character" << endl;
        kill(getpid(), SIGSTOP);
        // is not the size of a character
        return NULL;
    }

    bool nomos= (*_nummos>=*_maxnummos);
    bool nospc= (size > (*_size-(int) *_freememspcoffset));

    static int firsttime=1;
    if (firsttime) {
        if (mapMolist()<0)
            kill(getpid(), SIGSTOP);
        firsttime=0;
    }

    // Do the allocation part now.
    if (nomos || nospc) {
        // clean up some space for the Mo
        _mom->lock();
        while (_currentmonum < *_nummos) {
            cleanMo(_currentmonum);
            if (canDoAllocMo(_currentmonum, size) == 1) {
                // We have found a slot !
                _mom->unlock();
                return (_molist + _currentmonum++);
            }
            _currentmonum++;
        }
        // We are at the end of this extent.  Let Mom try another.
        _currentmonum = 0;
        _mom->unlock();
        return NULL;
    }
    else {
        // still some space in extent for a new mo
        return (allocInitialMo(size));
    }
}



int Extent::cleanMo(int block)
{
    // Inital conditions
    if (*_nummos < 2)  {
        return 1;
    }

    // Real code
    int cleanmo=((block+(*_nummos/2))%(*_nummos));

    Mo *mo;
    if (mo = _molist + cleanmo)
        mo->force_unreads();

    return 1;
}

int Extent::canDoAllocMo(int block, int size)
{
    // calculate Mo * from block
    Mo *mo= _molist + block;

    bool deleted_good_size=(mo->size()==-size);
    bool candelete=((_mom->isserver()==FALSE) &&
                    (mo->is_unreads()) &&
                    (mo->is_unpinned()));

    // decide if block has been deleted or can be deleted

    if (deleted_good_size) {              // mo has been deleted
        mo->set_size(size);
        return 1;
    }
    if (candelete) {                      // mo can be deleted
        delete mo;

        // check if deleted mo is the right size
        if (mo->size()!=-size) {
            return 0;
        }
        mo->set_size(size);
        return 1;
    }
    return 0;
}


int Extent::vsize()
{
    return (_PAGE_SZ+*_maxnummos*sizeof(Mo));
}
```