An RPC-Based External Memory Management Interface: Architecture

Randall W. Dean

Michelle Dominijanni

Open Software Foundation
Research Institute
1 Cambridge Center
Cambridge, MA 02142

Tel: (617) 621-7310, 621-7313
E-mail: rwd@osf.org, mmp@osf.org

Abstract. With the growing use of the Mach microkernel in both research and industry comes a demand for better performance. The Mach External Memory Management Interface has been modified to address shortcomings but its original asynchronous communication model has remained unchanged. During this time the performance of synchronous Mach IPC has improved dramatically while asynchronous Mach IPC has remained unimproved. This makes it worth while to consider fundamental changes to the EMMI. The existing asynchronous model also causes a great deal of complexity in the code that uses and implements the EMMI. By making the interface RPC-based, several components of Mach should become both faster and simpler. Several other projects are in progress that will enhance the performance of RPC, adding to the performance benefits. This document gives the highlights of the requirements and motivations discussed in preceding papers. It then describes the new EMMI interfaces, the enhancements in functionality it represents and some of the implementation details.[1]

1. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

1. Introduction

1.1 Requirements and Motivation

For a more complete description of the requirements, motivations and goals of this project, the papers *An RPC-Based External Memory Management Interface: Proposal*[1] and *An RPC-Based External Memory Management Interface: Requirements*[1] should be read. For a description of the current EMMI and a general description of the Mach interfaces, see *OSF Mach Kernel Interfaces[4].* This document assumes familiarity with these three papers, Mach and the EMMI in general. The remainder of this section will discuss some of the highlights of the requirements and motivations which have had the greatest effect on interface and architecture decisions.

1.2 Performance and RPC

One of the primary motivations for this project is improved performance, both immediate improvements by using more efficient communication methods, and future performance improvements by allowing the EMMI to leverage against future enhancements to RPC and IPC. Since the existing EMMI is asynchronous IPC based and the enhancements in communication are occurring for interfaces using RPC, it is necessary to convert the EMMI to RPC.

1.3 Stackable Pagers

The current implementation of transparent distributed shared memory, called XMM, is inserted into the system with source modifications to the kernel. XMM interposes itself between the kernel and pagers by explicitly adding code to the necessary kernel paths. The new EMMI should allow for the interposition of pagers such as XMM without kernel code modifications. This will entail the addition of a general pager interposition mechanism.

1.4 Pagers as First Class Objects

The current EMMI defines *memory objects* as first class objects. Memory objects have a privilege attribute. Pagers are a programming construct which has the receive right for the object. This design requires a minimum of two ports, one for the object, and a control port for the pager to communicate with the kernel about the object. It also requires that the privilege attribute be specified for each object at object initialization time.

Making pagers first class objects instead of memory objects should give us a number of improvements. Memory objects will be obtained from pagers, so they will inherit the privilege attribute. It will also no longer be necessary for memory objects to have any ports directly associated with them. A pager will have two ports, one for the pager to communicate with the kernel and one for the kernel to communicate with the pager. All manipulations of an object will occur by communications to or from its pager. Objects will be identified by an unique identifier.

1.5 Differentiated Kernel Calls and Pager Upcalls

It is not easy upon inspection of the current EMMI to determine which calls are made by the pager to the kernel and which are upcalls from the kernel to the pager. The new EMMI will clearly differentiate calls made by the pager to the kernel and upcalls from the kernel to the pager by using different naming conventions.

1.6 Trusted and Untrusted Pagers

The current EMMI is concerned about trusted and untrusted pagers in the context of whether or not to use the page in place optimization during pageout. The new EMMI must also be concerned about this as well as with the more general case of untrusted pagers not responding to RPCs.

1.7 Trusted and Untrusted Clients

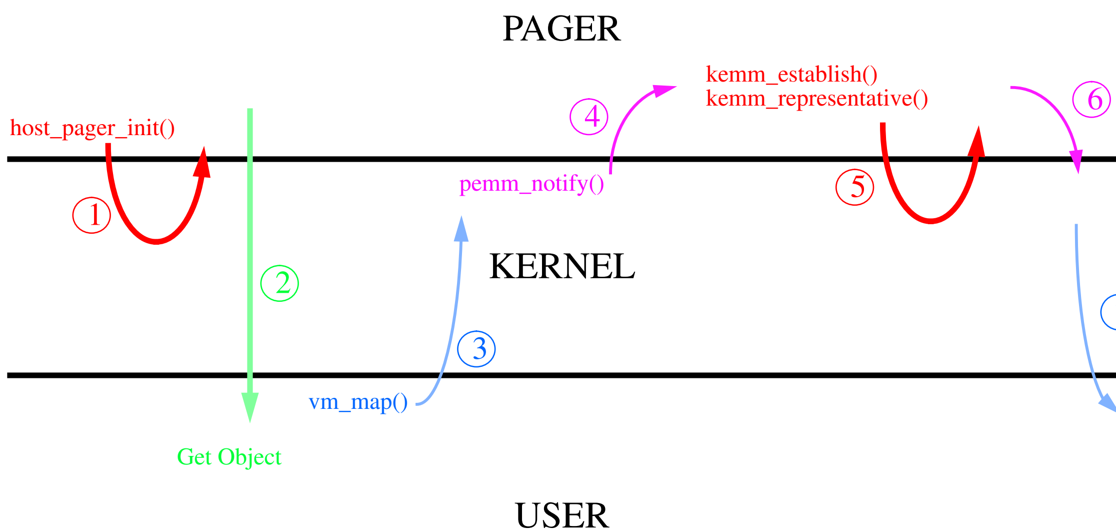
Distinguishing between clients a pager trusts and clients a pager does not trust can lead to an additional optimization with regard to port usage. If a client is not trusted, the pager must give the client a representative port to map an object. If the pager trusts the client to only map and access the object in a mutually agreed upon way, then it can instead give it a common port which identifies the pager and an opaque object identifier. This reduces the need for per-object ports and also potentially decreases the need to pass a port from the pager to the client for each object the client wishes to map.

1.8 Room for Future Expansion of Interface

The current EMMI has no room for extra parameters in any of the existing calls. Any expansion must take place by adding new calls, adding more parameters to new calls in an incompatible way, or overloading existing parameters. The new EMMI will leave room for future expansion within the defined interface.

2. Using the new EMMI

2.1 Pager and Object Initialization



This figure shows the interactions between pager and kernel which occur to initialize the pager and have a user map an object.

1) The first action a pager ever takes is to identify itself to the kernel with **host_pager_init()**.

2) The user gets either a *memory_object_representative* or the *memory_object* from the pager. This communication is not part of the EMMI and is therefore not specified.

3) The user **vm_map()**s the object.

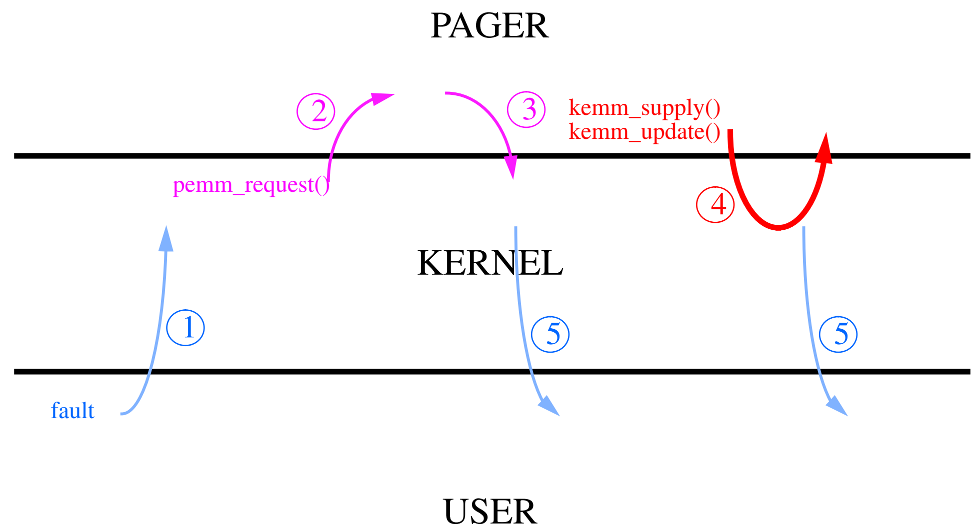
4) If the kernel does not know about the object or representative used for the map call, it contacts the pager with **pemm_notify()**.

5) If this is a new object the pager must then inform the kernel object with **kemm_establish()**. If it is a known object, but unknown representative then the pager uses **kemm_representative()**.

6) The pager returns from the **pemm_notify()** RPC. Any error from **kemm_establish()** or **kemm_representative()** is returned here.

7) The kernel completes the **vm_map()** call. Any error from the **pemm_notify()** is propagated to the user.

2.2 A User Pagefault



This figure shows how data is gotten from the pager.

1) The user first accesses data and a pagefault occurs.

2) If the data is not resident, or the lock value on the data does give sufficient access, the kernel will contact the pager with **pemm_request()**. This call combines the calls from the existing EMMI of **memory_object_data_request()** and **memory_object_data_unlock()**.

3) If the pager is immediately able to satisfy part or all of the request, it does so in the response to the **pemm_request()** RPC. If it is unable to satisfy any of the request immediately, it returns PAGER_QUEUED to the RPC.

4) If there is a portion of the request unsatisfied after the return from the **pemm_request()** RPC, then the pager is obligated to satisfy it in the future as the kernel will not make a request on the same region of an object until any pending request has been satisfied. To satisfy the request, the pager either uses **kemm_supply()** to supply data, or **kemm_update()** to unlock, zero fill, or indicate the data is in error.

5) The kernel satisfies the fault and returns to the user. This may occur upon completion of the original **pemm_request()** RPC if the necessary page request was satisfied as part of that RPC, or later when the pager supplies or updates the data.

3. Interface Specification

This section describes the proposed new EMM interface, the new host_pager interfaces and modifications and additions to existing subsystem interfaces. The EMMI interface is divided into upcalls made from the kernel to the pager which are all prefixed with **pemm_**, and calls made to the kernel by the pager which are prefixed with **kemm_**.

3.1 Host_pager Calls

```
kern_return_t host_pager_init(
        IN      host_pager_t              host_pager,
        IN      mach_port_t               authentication,
        IN      mach_port_t               pager_upcall,
        IN      mem_obj_rep_port_t        pager_map,
        IN      memory_object_flavor_t    flavor,
        IN      memory_object_info_t      attributes,
        OUT     mach_port_t               *pager_self);
```

Register this pager with a kernel. Use the host_control port for authentication for privileged access. Gets *pager_self* port back. If it already exists, returns success and existing *pager_self*. *Pager_map* port is for **vm_map_-trusted()**s from trusted clients. The *attributes* of type *flavor* are, if pager attributes, then the attributes of the pager, or, if object attributes, the default for any objects of this pager.

Returns:

• KERN_SUCCESS: returns **pager_self*

• KERN_FAILURE:

Existing Call:

None

```
host_pager_t host_pager_self()
```

Return the hosts host_pager port. This is the port for all **host_pager_init()** calls.

Returns:

• host_pager

Existing Call:

None

Additional calls will be added to this interface during the next phase of this project. These calls will include calls like **host_pager_interpose()** to allow a pager to interpose between the kernel and all of its pagers. This is how XMM would layer into the system. The interface will also specify an outcall to the interposition agent to involve it in the **pemm_notify()/kemm_establish()** dialog. This will work will occur when a more complete understanding of the needs of interposition agents such XMMare determined.

3.2 Pager Upcalls

The following calls are made from the kernel to the pager.

```
kern_return_t pemm_identify(
        IN       mach_port_t              default_paging_port,
        IN       host_name_port_t         host_name,
        OUT      mach_port_t              *pager_self);
```

The kernel *host_name* has been given *default_paging_port* with **vm_set_-default_memory_manager()**. The pager needs to supply the *pager_self* port associated with this port so that the kernel is able to page.

Returns:

- PAGER_SUCCESS: The port *pager_self* is valid and the pager is ready.
- PAGER_FAILURE: This is not a port associated with a pager on which default paging activity can occur.
- other: As returned by **host_pager_init()** from kernel. These also indicate failure.

Existing Call:

None

```
kern_return_t pemm_merge(
        IN       mach_port_t              pager_upcall,
        IN       pager_handle_t           memory_object_parent,
        IN       pager_handle_t           memory_object_backing,
        IN       existence_info_t         *existence_info
        IN       mach_msg_type_number_t   bitmap_count);
```

Merge *memory_object_backing* and *memory_object_parent* into a single object called *memory_object_parent*. If a page is present in both, discard the page in *memory_object_backing*. *Existence_info* describes all of the pages physically present in *memory_object_backing*. *bitmap_count* is the number of natural words in *existence_info*. The pager is responsible for invoking **kemm_update()** with *status->**merge*** set to move the necessary resident pages from the backing object to the parent object. The return from **pemm_merge()** has the effect of **kemm_destroy()** on the backing object. As with **pemm_return()**, where the object specified may be an unknown internal object, an unknown object may be given as *memory_object_backing* for this call. The pager should deal correctly with this possibility by creating the object for the duration of the call.
This call can only be made to a default pager.

Returns:

- PAGER_SUCCESS: The objects have been successfully merged.
- PAGER_FAILURE: Unable to merge the objects.

Existing Call:

None

```
kern_return_t pemm_notify(
        IN      mem_obj_rep_port_t      memory_object_representative,
        IN      pager_handle_t          memory_object,
        IN      host_name_port_t        host_name);
```

Notify the pager that an unknown representative or object has been used. If *memory_object_representative* is the *pager_upcall* port, then *memory_object* is the object. Otherwise, *memory_object_representative* is a representative port and *memory_object* is not defined. The pager must identify the object or representative to the kernel.

Returns:

- PAGER_SUCCESS: The object or representative is valid and the pager was able to inform the kernel.
- PAGER_INVALID_REPRESENTATIVE: The *memory_object_representative* port is not a valid representative port.
- PAGER_INVALID_OBJECT: The *memory_object* was invalid.
- other: Other failures as returned by **kemm_establish**() and **kemm_representative().**

Existing Call:

- **memory_object_notify()**

```
kern_return_t pemm_query(
        IN      mach_port_t             pager_upcall,
        IN      pager_handle_t          memory_object,
        IN      vm_offset_t             offset);
```

The kernel is requesting information about the page in *memory_object* at *offset*. The kernel needs to know if the pager has this data. This will only occur with the default pager.

Returns:

- PAGER_SUCCESS: The pager has the data.
- PAGER_DATA_UNAVAILABLE: The pager does not have this data.
- PAGER_FAILURE:

Existing Call:

None

```
kern_return_t pemm_request(
        IN      mach_port_t             pager_upcall,
        IN      pager_handle_t          memory_object,
        INOUT   vm_offset_t             *offset,
        INOUT   vm_size_t               *size,
```

```
OUT      vm_offset_t               *data,
OUT      vm_size_t                 *data_count,
OUT      boolean_t                 *deallocate
INOUT    data_info_t               *status);
```

The kernel is requesting valid data at **offset* of size **size* and protection *status->**protection***. *status->**kernel_copy*** indicates whether the kernel currently has data and needs the data unlocked, or if the kernel needs the data supplied. *status->**overwrite*** being set, *status->**kernel_copy*** not being set and a desired access of write indicates that the kernel wishes to overwrite the existing data. The pager need not supply the data, since the kernel will just overwrite it. They pager may choose to discard the data it has for this page since it is no longer valid.

The return of PAGER_QUEUED indicates that the pager was unable to immediately satisfy any part of the request and subsequent **kemm_supply()** and/or **kemm_update()** calls will satisfy the request. All OUT parameters have no defined meaning with the return of PAGER_QUEUED. PAGER_SUCCESS indicates that a portion or all of the request has been satisfied and information about it is available in the return parameters.

If the pager is returning data in return to the **pemm_request()** RPC, then **data_count* will be non-zero, and indicate the size of the data be returned in **data* with **deallocate* indicating whether to deallocate the data from the pagers address space, **offset* specifying the beginning offset of the data and **status* fields containing further information as specified in the **kemm_supply()** call. **data_count* being set makes the return from the RPC the equivalent to **kemm_supply** without the available return information.

If **data_count* is zero, then this return is the equivalent of the **kemm_update()** RPC without the lock semantics associated with the return from **kemm_update().** **offset* and **size* indicate what range of the object the return command satisfies. *status->**error*** indicates a data error and *status->**data.page_error*** contains the error. *status->**unavailable*** will be set if the kernel did not have the data and the pager does not either and the object is internal. *status->**zero_fill*** is set under the same circumstances if the object is external. If *status->**error*** is not set, *status->**lock_value*** is valid. If *status->**lock_value*** does not give the access required, then *status->**data.page_error*** will contain an error code from the pager regarding the failed access. This error is propagated to the user.

Returns:

- PAGER_SUCCESS: The request for pages at **offset* of size **data_count* or **size* has been satisfied via the information in **status.*
- PAGER_QUEUED: request has/is being dealt with via other means. No return parameters contain any defined values.

• PAGER_FAILURE:

Existing Call:

• **memory_object_data_request()**

• **memory_object_data_unlock()**

```
kern_return_t pemm_return(
        IN      mach_port_t             pager_upcall,
        IN      pager_handle_t          memory_object,
        IN      vm_offset_t             offset,
        IN      pointer_t               data,
        IN      vm_size_t               data_count,
        IN      data_info_t             status);
```

The kernel is returning data to the pager. The following fields are valid in *status: **kernel_copy, dirty, flushed***, ***lock_value*** and ***precious.*** If ***kernel_-copy*** is set, ***lock_value*** will also be valid. The *memory_object* may be unknown to the pager if it is acting as a default pager. If the object is an internal object and it is a new, then the pager must do whatever initialization is necessary and accept the pageout. If the object is unknown and the object is not internal, this is an error and the pager should return PAGER_-FAILURE.

A PAGER_SUCCESS return to the RPC indicates that the pager is now able to deal with pagein requests on the same data. The other action the pager must eventually take is to deallocate the memory. For a privileged pager, this is the hint to a kernel using the clean in place optimization, to remove the page from the original object.

Returns:

• PAGER_SUCCESS: The pager has received the pageout request.

• PAGER_FAILURE:

Existing Call:

• **memory_object_data_return()**

```
kern_return_t pemm_terminate(
        IN      mach_port_t             pager_upcall,
        IN      pager_handle_t          memory_object);
```

The object has been terminated. The kernel will no longer accept messages regarding this *memory_object* nor will it send any. This call will not be made until all pending RPCs on this object have completed.

Returns:

• PAGER_SUCCESS: success.

• PAGER_FAILURE:

Existing Call:

- **memory_object_terminate()**

```
kern_return_t pemm_update(
        IN      mach_port_t             pager_upcall,
        IN      pager_handle_t          memory_object,
        IN      vm_offset_t             offset,
        IN      vm_size_t               data_count,
        IN      data_info_t             status);
```

Update use information about a range of pages. The events which drive this upcall are determined by the *memory_object*'s attributes of MEMORY_-OBJECT_UPDATE_INFO flavor. State changes which can generate upcalls are *status.**dirty*** and *status.**flushed***. The *status* fields ***kernel_copy***, ***dirty***, ***flushed***, ***error***, ***precious***, and ***lock_value*** are always valid when this upcall occurs.

*status->**flushed*** events occur when pages are flushed from the kernel and no **pemm_return()** occurs as a result. *status->**dirty*** events occur the first time a page becomes dirty.

Returns:

- PAGER_SUCCESS: success.
- PAGER_FAILURE:

Existing Call:

None

3.3 Calls to Kernel

The following are calls made by the pager to the kernel.

```
kern_return_t kemm_change_attributes(
        IN      mach_port_t             pager_self,
        IN      pager_handle_t          memory_object,
        IN      memory_object_flavor_t  flavor,
        IN      memory_object_info_t    attributes);
```

Change the attributes of the specified *memory_object*. If the changing of an attribute results in object termination, the reply to this will be received after the **pemm_terminate()** RPC completes. A *memory_object* of MEMO-RY_OBJECT_PAGER will change the default attributes of that *flavor* for the pager.

Returns:

- KERN_SUCCESS:
- KERN_FAILURE:

Existing Call:

- **memory_object_change_attributes()**

```
kern_return_t kemm_destroy(
        IN      mach_port_t             pager_self,
        IN      pager_handle_t          memory_object);
```

The pager wants *memory_object* destroyed. This is the last interaction on this *memory_object* which can occur. No further messages to the kernel or upcalls from the kernel are valid, including **pemm_terminate**().

Returns:

- KERN_SUCCESS:
- KERN_FAILURE:

Existing Call:

- **memory_object_destroy()**

```
kern_return_t kemm_end(
                mach_port_t             pager_self);
```

This is no longer a registered pager. Destroy all objects associated with it. This has the same semantics with regard to the objects backed by this pager as the pager iterating over all of its objects and calling **kemm_destroy()** on them.

Returns:

- KERN_SUCCESS: success.
- KERN_FAILURE:

Existing Call:

None

```
kern_return_t kemm_establish(
        IN      mach_port_t             pager_self,
        IN      ledger_port_array_t     ledgers,
        IN      mem_obj_rep_port_t      memory_object_representative,
        IN      pager_handle_t          memory_object,
        IN      representative_attr_t   representative_attributes,
        IN      memory_object_flavor_t  flavor,
        IN      memory_object_info_t    behavior,
        IN      mach_msg_type_number_t  behavior_count);
```

Establish new object optionally specifying a representative.

Returns:

- KERN_SUCCESS: object and representative set up
- KERN_OBJECT_EXISTS: object exists

- KERN_REPRESENTATIVE_MISMATCH: representative exists for another object
- KERN_FAILURE:

Existing Call:

- **memory_object_establish**()

```
kern_return_t kemm_get_attributes(
        IN      mach_port_t                 pager_self,
        IN      pager_handle_t              memory_object,
        IN      memory_object_flavor_t      flavor,
        OUT     memory_object_info_t        *attributes,
        INOUT   mach_msg_type_number_t      *size);
```

Get *memory_object* attributes of type *flavor*. A *memory_object* of MEMORY_OBJECT_PAGER will get the default attributes of that *flavor* for the pager.

the pager.

Returns:

- KERN_SUCCESS:
- KERN_FAILURE:

Existing Call:

- **memory_object_change_attributes()**

```
kern_return_t kemm_representative(
        IN      mach_port_t                 pager_self,
        IN      mem_obj_rep_port_t          memory_object_representative,
        IN      pager_handle_t              memory_object,
        IN      representative_attr_t       representative_attributes);
```

Specify representative for an object. This can be done at any time or in the response to **pemm_notify**(). If it is unsolicited, the representative information may just be discarded like any representative information without an explicit reference.

Returns:

- KERN_SUCCESS: representative set up
- KERN_NO_OBJ: no such *memory_object* exists
- KERN_FAILURE:

Existing Call:

- **memory_object_representative()**

```
kern_return_t kemm_supply(
        IN      mach_port_t                 pager_self,
        IN      pager_handle_t              memory_object,
        INOUT   vm_offset_t                 *offset,
```

```
        INOUT    pointer_t                *data,
        INOUT    vm_size_t                *data_count,
        IN       boolean_t                deallocate,
        IN       data_info_t              status);
```

The pager is supplying **data* to the kernel for *memory_object* of size **data_count* at **offset* and with access of *status->**lock_value***. *status->**precious*** indicates whether the page is precious. If *deallocate* is set, the memory is deallocated from the pager address space when the message is created. On any return except KERN_SUCCESS **offset* indicates the offset into the object at which the error occurred. If an error occurs, all memory preceding the error page was accepted by the kernel. The kernel does not attempt to process any of the data after the error page. If *status->**precious*** is set and *deallocate* is true, on error, the unprocessed data is returned at **data*.

Returns:

- KERN_SUCCESS: Data successfully supplied
- KERN_MEMORY_PRESENT: The data at **offset* was already present. All data preceding it was successful and following it was untried.
- KERN_FAILURE:

Existing Call:

- **memory_object_data_supply()**

```
kern_return_t kemm_update(
        IN       mach_port_t              pager_self,
        IN       pager_handle_t           memory_object,
        IN       vm_offset_t              offset,
        IN       vm_size_t                size,
        IN       data_info_t              status);
```

This call updates information about a range of pages in the kernel. This is very similar to KERN_SUCCESS being returned with **pemm_request()**. The fields in *status* indicate the new status of the pages. If ***merge*** is set, no other fields have meaning. The ***merge*** field indicates the pages should be moved from this object, which is the *memory_object_backing* of a **pemm_merge()** call to the parent object. If a **pemm_merge()** call is not currently in progress on this object, the setting of this bit is an error which will result in no action being taken. ***flushed*** requests that the kernel flush the data. ***error*** indicates that the pages are now in error and ***data.page_error*** contains the error code to associate with the pages. Any data currently present should be discarded. ***data.should_return*** tells whether to return the data to the pager or not when ***error*** is not set. ***precious*** has no side effect other than setting the precious bit on the page. If there is no data present, this has no effect. ***zero_fill*** causes the kernel to zero fill the page but only is valid to specify on an external object. If data was present, it is lost.

unavailable is specified for an internal object if the pager does not have the data and is trying to satisfy a **pemm_request()** call. ***lock_value*** indicates the new lock value for the data. This field is valid in all cases except ***flushed***, ***merge*** and ***error.***

Returns:

• KERN_SUCCESS: The requested action has completed

• KERN_FAILURE:

Existing Call:

• **memory_object_data_error()**

• **memory_object_data_unavailable()**

• **memory_object_lock_request()**

3.4 Other Interfaces

```
kern_return_t vm_map_trusted(
        IN      task_port_t             target_task,
        INOUT   vm_address_t            *address,
        IN      vm_size_t               size,
        IN      vm_address_t            mask,
        IN      vm_allocate_t           anywhere,
        IN      mem_obj_rep_port_t      memory_object_representative,
        IN      pager_handle_t          memory_object,
        IN      vm_offset_t             offset,
        IN      boolean_t               copy,
        IN      vm_prot_t               cur_protection,
        IN      vm_prot_t               max_protection,
        IN      vm_inherit_t            inheritance);
```

vm_map_trusted() is exactly the same as the existing **vm_map()** call with the exception of the additional pager_handle_t argument. The argument *memory_object_representative* must either be a representative port, or a *pager_map* port. If it is the former, the meaning of the argument *memory_-object* is undefined. If it is the latter, then *memory_object* is the opaque handle for the object. The semantics of this call are otherwise the same as the existing **vm_map()** call.

```
kern_return_t vm_object_close(
        IN      object_handle_port_t    handle);
```

Release object reference and invalidate handle.

```
kern_return_t vm_object_open(
        IN      mem_obj_rep_port_t      memory_object_representative,
        IN      pager_handle_t          memory_object,
        IN      vm_prot_t               protection,
        OUT     object_handle_port_t    *handle);
```

Acquire reference on an object and make sure it is valid and initialized. This gives a handle for vm_object_read and vm_object_write.

Returns:

- KERN_SUCCESS:
- KERN_FAILURE:

```
kern_return_t vm_object_read(
        IN      object_handle_port_t        handle,
        IN      vm_offset_t                 offset,
        IN      vm_address_t                address,
        IN      vm_size_t                   size);
```

Effectively bcopy from object.

```
kern_return_t vm_object_write(
        IN      object_handle_port_t        handle,
        IN      vm_offset_t                 offset,
        IN      vm_address_t                address,
        IN      vm_size_t                   size
        IN      object_write_flag_t         flag);
```

Effectively bcopy to object.

```
kern_return_t vm_set_default_memory_manager(
        IN      host_priv                   host_priv_t,
        INOUT   mach_port_t                 *default_manager);
```

The *default_manager* port can be the already initialized *pager_self* port of a pager, or the *pager_upcall* port. If the kernel recognizes the port as identifying a pager, no upcall will take place. If it does not, it calls **pemm_identify()** on the supplied port. It does not return until it successfully sets up a link with the pager. The out parameter is the *pager_upcall* port of the old default pager.

Returns:

- KERN_SUCCESS: Now using new default pager.
- KERN_FAILURE: Unable to initialize new pager.

4. Data Structures

4.1 data_info_t

```
typedef struct data_info {
        unsigned int
                overwrite:1,        /* Data is being overwritten */
                dirty:1,            /* Data is or is becoming dirty */
                flushed:1,          /* Page is being or should be flushed*/
                error:1,            /* The page is in error */
```

```
                unavailable:1,     /* The pager does not know about this
                                           data*/
                zero_fill:1,       /* This page should be zero filled*/
                merge:1,           /* Move these pages to parent */
                precious:1,        /* This data is precious */
                lock_value:4;      /* Current or new lock value */
                kernel_copy:1,     /* The kernel is keeping a copy */
                protection:4,      /* The desired access to the data */
        union {
                kern_return_t                  page_error;
                memory_object_return_t         should_return;
        } data;
} data_info_t;
```

In this document this data structure is defined to contain bit fields. This is done primarily for ease of description. The true definition of the first word of the structure is a unsigned int where the least significant bit (0x00000001) defines the field ***overwrite***, the next bit (0x00000002) defines ***dirty***, etc. Whether to actually use bitfields or an unsigned integer and masks is a implementation specific decision likely to be based on how the compiler generates code for the given architecture.

The data_info_t structure shows up in most EMMI calls as the field *status*. The fields ***dirty***, and ***flushed*** are both status bits set when an **pemm_update()** event occurs of that type. The fields ***flushed***, ***error***, ***unavailable***, ***zero_fill***, ***merge***, ***precious***, and ***lock_value*** are all used as commands from the pager to the kernel to set the state of a page. The fields ***overwrite, dirty***, ***precious***, ***lock_value***, ***kernel_copy***, and ***protection*** are set by the kernel to either indicate current or desired status for a page.

data.page_error is a status field which is only valid when ***error*** is set. The field ***data*** is otherwise interpreted as ***data.should_return*** whenever a command is being given from the pager to the kernel.

4.2 pager_handle_t

```
typedef unsigned int pager_handle_t;
#define MEMORY_OBJECT_PAGER 0
```

Pager_handle_t is the type of all memory_objects. The assigning of values for memory_objects depends on whether the object is an internal or external object. External objects are named by their pagers. Internal objects are named by the kernel. The memory_object MEMORY_OBJECT_PAGER is reserved to specify the pager, and not one of its object when using the calls **kemm_change_attributes()** and **kemm_get_attributes()**.

4.3 Existence Bitmap

The existence_info_t structure is passed in **pemm_merge()**.

```
typedef struct {
        vm_offset_t        starting_offset;
        int                size;
        char               bitmap[1];
} existence_info_t;
```

What is passed to **pemm_merge()** is a vector of these structures. The last structure is determined by having a size value of zero. Each structure defines a section of the object. Any section not described does not contain any resident pages. ***starting_offset*** is the starting point of the object which ***bitmap*** describes. ***size*** is the number of bits which are valid in ***bitmap***. ***bitmap*** is in fact an arbitrarily large array.

4.4 Representative Attributes

```
typedef struct {
        security_id_t     security_id;
        vm_prot_t         max_protection;
        vm_offset_t       starting_offset;
        vm_offset_t       ending_offset;
} representative_attr_t;
```

This structure describes the attributes of an object representative. ***security_id*** is the security id a client must have to map the object. ***max_protection*** is the maximum allowable access the mapper may have. ***starting_offset*** and ***ending_offset*** describe the range of the object which the representative is allowed access to. An ***ending_offset*** of zero indicates the end of the object.

4.5 Object Attributes

```
#define MEMORY_OBJECT_INFO_MAX                (1024)
typedef int       *memory_object_info_t;
typedef int       memory_object_flavor_t;
typedef int       memory_object_info_data_t[MEMORY_OBJECT_INFO_MAX];

#define MEMORY_OBJECT_BEHAVIOR_INFO           10
#define MEMORY_OBJECT_PERFORMANCE_INFO        11
#define MEMORY_OBJECT_UPDATE_INFO             13
#define MEMORY_OBJECT_COHERENCY_INFO          14

struct memory_object_behave_info {
        memory_object_copy_strategy_t         copy_strategy;
        boolean_t                             temporary;
        boolean_t                             invalidate;
        boolean_t                             write_completions;
};

struct memory_object_perf_info {
        vm_size_t                             cluster_size;
        boolean_t                             may_cache;
};

struct memory_object_update_info {
        boolean_t                             flushed;
        boolean_t                             dirty;
};

struct memory_object_coherency_info {
        coherency_t                           coherency;
};
```

4.6 Object_write_flag_t

```
typedef struct object_write_flag {

} object_write_flag_t;
```

This structure's definition will be determined later as the needs of the **vm_object_** interfaces become better understood.

4.7 Object_handle_port_t

```
typedef mach_port_t         object_handle_port_t
```

A object_handle_port_t is the OUT parameter of **vm_object_open().** Subsequent **vm_object_{read,write,close}** calls require this parameter.

4.8 Host_pager_t

```
typedef mach_port_t         host_pager_t
```

This port is a host special port which is used for initiatializing all pagers. It will also be used for the pager interposition interface to be specified at a later date.

5. Enhancements and Implementation Details

5.1 Performance

5.1.1 Pagefault Handling

Table 1: Page Fault Handling

Communication	Data Supply		Zero Fill	
	Time (µs)	Percent Speedup	Time (µs)	Percent Speedup
Current Mach IPC	950	0	830	0
RPC	790	17	670	20
Migrating RPC	724	24	604	27
Short Circuited RPC	585	38	585	30

We have run some micro-benchmarks on a 50MHz 80486 machine to determine the current performance of supplying and zero-filling data as the result of a user pagefault. From the results of the micro-benchmarks shown in the column labeled "Current Mach IPC", we have extrapolated times for the new RPC based EMMI for three different RPC methods. The first is using the existing MACH IPC to do RPC. Since RPC has been optimized, there is a noticeable improvement. The second type of RPC is the migrating RPC being used by the MK6 projects and the Real-Time projects. The final RPC examined is that being used by collocated servers. These extrapolations show the minimum possible improvements possible by switching the EMMI to a RPC communication paradigm.

5.1.2 Paging Anonymous Memory

The paging of anonymous memory to the default pager is the area where we are most likely to see the benefits of these proposed interfaces for the EMMI. When paging to a remote node, the removal of the **memory_object_create()** call the first time an internal object is paged, will greatly decrease the setup costs associated with paging. This combined with the removal of the back channel discussed in section 6.2 should have a noticeable effect on paging performance.

Paging to a local default pager will not only benefit from reduced setup costs, but a reduction in communication costs for each pageout message. The pageout overhead will decrease in an absolute fashion similar to that shown in Table 1, "Page Fault Handling." Percentage wise, the overhead should decrease more since there is less non-communication overhead associated with pageouts.

5.2 Removal of Back Channel in Pageout Path

The current pageout path requires that the kernel first do a **memory_object_create()** to the pager to inform the pager of the existence of the internal object, then start paging the data out. The **memory_object_create()** call moves a receive right from the kernel to the pager for the object. With NORMA IPC and paging to a remote node, this requires a message from the node on which the pager resides to be received by the node paging to pull the receive right over. Since the node paging is short of memory, it may not be able to receive a message. This back channel would result in deadlock if it was not special cased.

The new EMMI divides the object name space in two with the **internal** bit in the pager_handle structure. The naming of objects with the internal bit set is controlled by the kernel. Other than notifying the pager of the object name, the only other information contained in the **memory_object_create()** call is vm_page_size for the kernel and object size. The former can be determined at pager initialization time. The latter is not necessary for correct behavior. The result is that no initialization message needs to be sent before pageout can occur. When the pager gets a pageout message for an internal object it does not already know about, it does whatever initialization it needs to at that time. This results in the elimination of the paging back channel and a speed up in the pageout path.

5.3 Default Pager Support for Object Merging

In the current system, when the kernel attempts to collapse a shadow chain, it is only able to do so if the backing object does not have a pager. If an object cannot be collapsed because of the existence of a pager, the kernel attempts to bypass the backing object. To bypass the object, all of the pages resident in the object or paged out must either be resident or paged out in the parent object. We have added a new pager upcall, **pemm_merge()** to allow objects to be merged even if there is a pager and regardless of whether all the pages yet reside in the parent object.

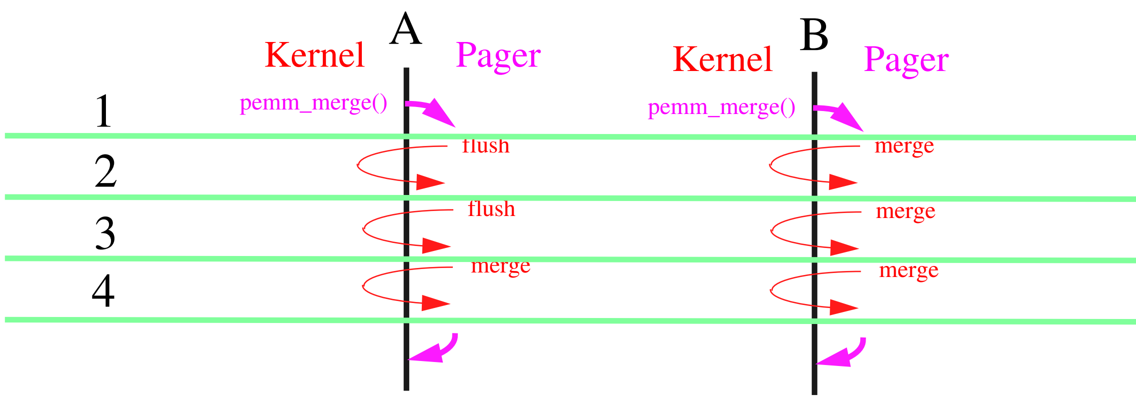
5.3.1 Case Study

When collapsing shadow chains, the most difficult case arises when both the backing and parent objects have pagers and the backing object has resident data. As for all cases, any pages which are resident in both the backing and parent objects can be discarded from the backing object. The kernel then creates an existence bitmap for the backing object. This bitmap plus the names of the two object is passed to the pager with **pemm_merge()**. The pager then merges the data on the backing store and moves the necessary resident pages from the backing object to the parent as described in the **pemm_merge()** specification.

If the backing object has resident data but no pager, **pemm_merge()** is still called as in the first case. If there is a pager for the backing object but none for the parent object, the kernel can just change the pager to back the parent object and resolve the merge itself. All of the remaining possibilities can also be resolved by the kernel without contacting the pager.

5.3.2 Paging During a Merge

It is possible for pageouts to occur during the middle of a merge operation. After sending a **pemm_merge()** and before the pager responds to the RPC, it is possible for the kernel to pageout data in either of the two objects. The pager must be able to handle this possibility.



This figure shows two possible methods a pager could use to satisfy a **pemm_merge().** In case A, the pager flushes the data which should be flushed with **kemm_update()** and then invokes **kemm_update()** over the whole object to merge the remaining pages.

In case B, the pager steps through the object merging the necessary pages with **kemm_update()** and does not worry about the extraneous ones since they will be discarded when the **pemm_merge()** RPC completes and the backing object is destroyed.

Future versions of this document will analyze the effects of paging activity at each of the eight possible locations in the above figure.

5.4 Support for Querying Pager

To correctly handle the pushing of a page to a copy object, the existing EMMI must send a **memory_object_data_initialize()** message to the copy object's pager. This message contains the data being pushed. The pager then discards the data if it already has the page or keeps it if it does not. The kernel must take this action because it does not know what is on backing store and has no way of determining this. To avoid some of these messages, the existing kernel has a bitmap which keeps track of what is on backing store. This requires data structure space which may never be used, and which cannot solve the entire problem since it is of a fixed size and cannot handle arbitrarily large objects. Given that we now have a much faster mechanism for communicating with the pager, it makes sense to just have the kernel query the pager whether the data is on backing store before sending it. For co-located default pagers, this will be a tremendous space savings.

5.5 Pager Upcalls to Track Page Usage Information

One of the shortcomings of the existing EMMI is that it does not have a mechanism for notifying the pager of an object about state changes in the pages of that object. A pager which backs the memory used to implement a file system buffer cache might want to know if its pages are resident in memory or if they have been discarded. If the page is resident, then no paging activity would have to occur to reuse that page. If it has been discarded because the kernel has it marked clean, and the buffer cache manager used it, it would require a pagein before it could be overwritten. The first upcall we have added is the ***overwrite*** event which will allow the kernel to just allocate a new zero filled page and overwrite it. The pager is notified so it can potentially remove any data from backing store, or, if implementing distributed shared memory, invalidate copies of the pages on other kernels before replying.However, using this call alone still requires the allocation of a new page in the kernel. To avoid this, there is a new ***flushed*** event which notifies the pager that a clean page has been flushed. This event allows the manager to pick a different, still resident page to reuse. Another event which the research community has shown would be useful to be notified of is the ***dirty*** event. This occurs the first time a page goes from clean to dirty. All of the events can be asked for or not as the pager desires.

5.6 Direct Access to Memory Objects Without Mapping

This interface will be more completely specified in the future. The purpose of these additions is to allow direct access to memory_objects without needing to map the data into the accesser's address space. The purpose of requiring the allocation of a handle with **vm_object_open()** is to allow an explicit reference to be taken out on an object. This particular feature is important when the pager wishes to control object caching behavior when their are currently no clients instead of having the kernel control the caching.

5.7 Removal of Sequence Numbers

All pagers must now be multithreaded because of the overlapping nature of the new RPC based EMMI. A **kemm_update()** which causes a page to be flushed and sent to the pager requires a **pemm_return()** call to begin and complete before the response to the **kemm_update()** call. This requires that the pager have multiple threads to handle a given object. In the existing EMMI sequence number interfaces were needed to avoid race conditions. Since all interactions between the kernel and pager are now RPCs, it is possible to avoid these race conditions by not allowing certain events to take place until a pending RPC has completed.

One good example is **pemm_return()** and a subsequent **pemm_request()** on the same data. In the existing EMMI, a **memory_object_data_return()** could be followed immediately by a **memory_object_data_request()** for the same page. If the thread handling the request raced ahead of the return handling, then the kernel could get stale data. The existing EMMI solves the race by tagging each upcall with a sequence number which allows the pager to avoid the race. In the new EMMI we define **pemm_request()** on a page as not being initiated until the return from the **pemm_return()** RPC. By analyzing the other race conditions and defining similar semantics as are defined to avoid this race, we eliminate the need for the sequence number interfaces.

5.8 VM Pageout Thread

In the current code base where Mach IPC is used as the primitive for building RPC, untrusted pagers can be dealt with by sending the first half of the RPC as an asynchronous message and having a dedicated thread for dealing with reply messages.

5.9 Migrating Threads

With RPC being a primitive built on migrating threads, it will be necessary to have the pager donate a thread context for these RPCs to take place in.

References

[1] Dean, R.W. and Dominijanni, M., An RPC-Based External Memory Management Interface: Proposal, OSF RI internal document, November 1993.

[2] Dean, R.W. and Dominijanni, M., An RPC-Based External Memory Management Interface: Requirements, OSF RI internal document, February1994.

[3] McNamee, D., Supremo, personal communication.

[4] Open Software Foundation and Carnegie Mellon University, OSF Mach Kernel Interfaces, OSF RI internal document, May 1993.

[5] Open Software Foundation Research Institute, Kernel to Kernel Transport Interface for the Mach Kernel, OSF RI internal document, January 1994.

[6] Open Software Foundation Research Institute, NORMA IPC Version Two: Architecture and Design, OSF RI internal document, January 1994.