# Mach Tutorial

David L. Black

February 1991

Copyright ©1990 by David L. Black,
Avadis Tevanian, Jr., and Michael Wayne
Young
All Rights Reserved.

# Virtual Memory Management in Mach

# Mach VM: Key Features

---

- *Architecture independence*: easily portable to paged architectures
- *Multiprocessor/distributed system support*: supports current and future architectures
- *Advanced functionality*: shared memory, mapped files, etc.
- *High performance*: outperforms several machine dependent designs

# Mach VM Support

---

Interface

- portable – simple paged address space model
- flexible – sparse, virtual address spaces
- sharing – within a task, and between related tasks
- extensible – user-defined memory objects

Implementation

- lazy evaluation
- copy-on-write virtual copy operations
- architecture-independent data structures
- easily portable - minimal hardware support required

## Basic VM Design

---

Needs to be portable to wide variety of hardware

Logical (machine-independent) representation

- Task virtual memory *address maps*
- Virtual memory *objects*
- Resident *pages* – not necessarily hardware page size

Architecture-dependent subsystem

- Keep the interface simple
- Merely make the MMU work
- May have incomplete information

# Entry Lookup

---

*vm_map_lookup_entry*: lookup an address in a map, returning a pointer to an entry.

Algorithm:

- Start at saved hint.
- Follow linked list until region found.

# Map space allocation

---

*vm_map_find*: finds free address space

Algorithm:

- Use free space hint when possible.
- Start at specified address (using lookup).
- Follow list until finding an open slot.
- If slot found, insert new entry into list (extending previous entry if possible).

# Other Map Manipulations

---

*vm_map_remove*

*vm_map_copy*

*vm_map_protect*

*vm_map_inherit*

All operate on a range of addresses.

First lookup start address.

Scan each region in the list (until end) performing necessary operation.

Split (and merge) entries whenever necessary.

# VM Objects

---

Each VM object represents a single source of virtual memory.

Resident pages cache that memory in physical memory.

Remainder is always available from object's *memory manager*.

No memory manager is recorded for zero-filled memory initially. Default memory manager will provide backing storage when pageout becomes necessary.

All zero-filled memory is created equal... sequentially allocated memory extends objects (instead of creating new ones).

# Efficient Virtual Memory Copy

---

Copied data is often only read, not written

Share physical memory read-only until written, then make copy

Copy-on-write algorithm:

- Copy object reference to new map
- Mark both map entry references *copy-on-write*
- Reprotect pages in hardware to prevent writing

When write fault occurs:

- Make new object to hold changed pages
- Keep old object reference for unchanged pages

# Resident Pages

---

Physical memory is just a cache of virtual memory

Page belongs to just one object

Associative store: hash to resident page by object, offset (called the *virtual-to-physical table*)

Also need to maintain:

- All pages within one object (to ease deallocation, virtual copy)
- Pageout information
  - Modified and referenced bits
  - Free queue, or LRU pageout queue

Page structure is a multi-threaded list

Wire operation faults in page; removes from queues.

# Page Fault Algorithm

---

Look up fault address in map to find object. Create shadow object if necessary.

Get the specified page in resident memory (next slide)

Make copy if copy object doesn't have one and page is being written.

Verify that we still want this page; if not, start all over

Enter page into physical map and return

# Page Fault – How to Find a Page

---

Have an object/offset description.

Until we find the page:

- Look for a resident page in this object
- If we can't find one:
  - Allocate new page with appropriate object/offset; mark it busy
  - Ask the memory manager for this object to retrieve the data
    - If we still don't find it, try loop again with shadow object
    - If there is no shadow object, zero-fill the page

# Pageout Algorithm

---

Three queues:

- Active list – pages that have been mapped, in LRU-like order
- Inactive list – pages with reference bit cleared, staged for pageout.
- Free list – pages with no mappings; available for use

Free clean pages from the *inactive* list

Start cleaning dirty pages from the *inactive* list

Replenish the *inactive* list by drawing from *active* list

Reference bit need not be implemented as such in pmap module

# Virtual Memory Cache

---

All pages in memory represent a VM cache of secondary storage.

Objects may be specified as *persistent*:

- When an object is no longer referenced, its pages are not freed.
- Pages are placed on inactive queue (remembered in hash table).
- Kernel maintains a LRU list of unreferenced objects still cached

Major use - objects representing virtual memory obtained from files

# Locks

---

The following locks are used:

- *map locks*: protects address maps
- *object locks*: protects each object
- *hash table bucket locks*: locks object/page hash table
- *busy pages*: long term per-page locks

Notes:

- Address maps have subsidiary locks for reference count and hints.
- Currently only a single hash table bucket lock - may change in future.

# External Memory Management in Mach

# Exporting Memory Management

---

Map objects into user address spaces

User-level server manages backing storage

Kernel manages main memory as cache of user data

# Benefits

---

Flexibility in managing data

Improved performance from better sharing and caching

Better structuring of kernel and applications

Allows virtual memory to be implemented on a communication base – appropriate to some multiprocessors

# Applications

---

Typical Operating System Facilities

- Mapped files
- Shared state mapped as objects
- Network paging
- Network shared memory

Database Management

- Special backing storage management
- External consistency constraints

Process Migration

- Copy-on-reference
- Provide-in-advance

Relaxed Consistency on Shared Data

# Interface Organization

---

Memory objects

- Mapped into user address space(s)
- Represented by a port if user-managed
- Port used by kernel to access and store object data

Interfaces

- Memory Manager to Kernel
- Kernel to Memory Manager

Asynchronous

- Provide data in advance
- Delay transfer to backing storage
- Allow flexibility and concurrency

Multiple kernels may use same object

# Interface Description

---

Client - Map memory object

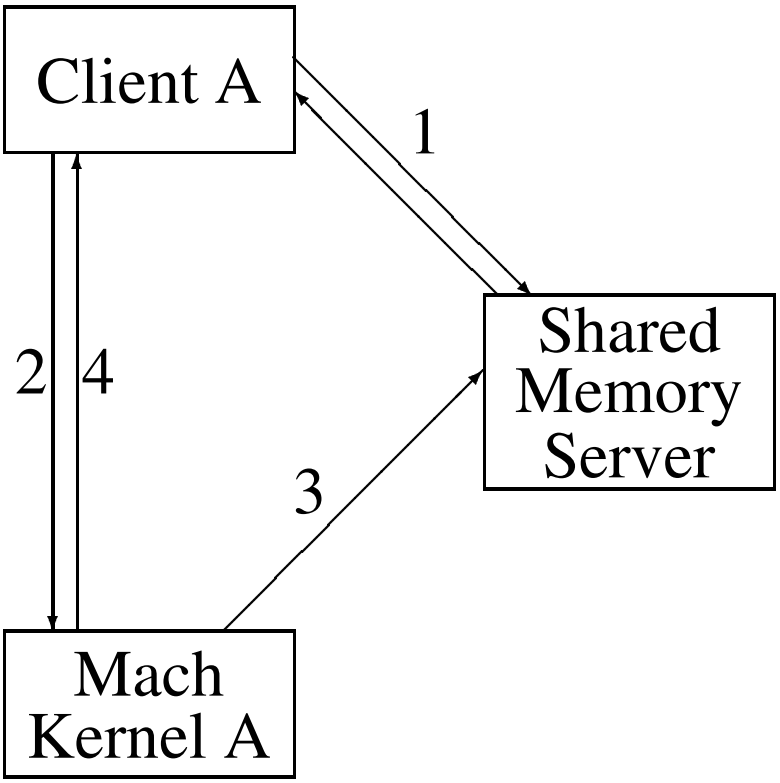
Kernel to Memory Manager

- Initialize memory object
- Request page of data
- Write back changed data page
- Terminate

Memory Manager to Kernel

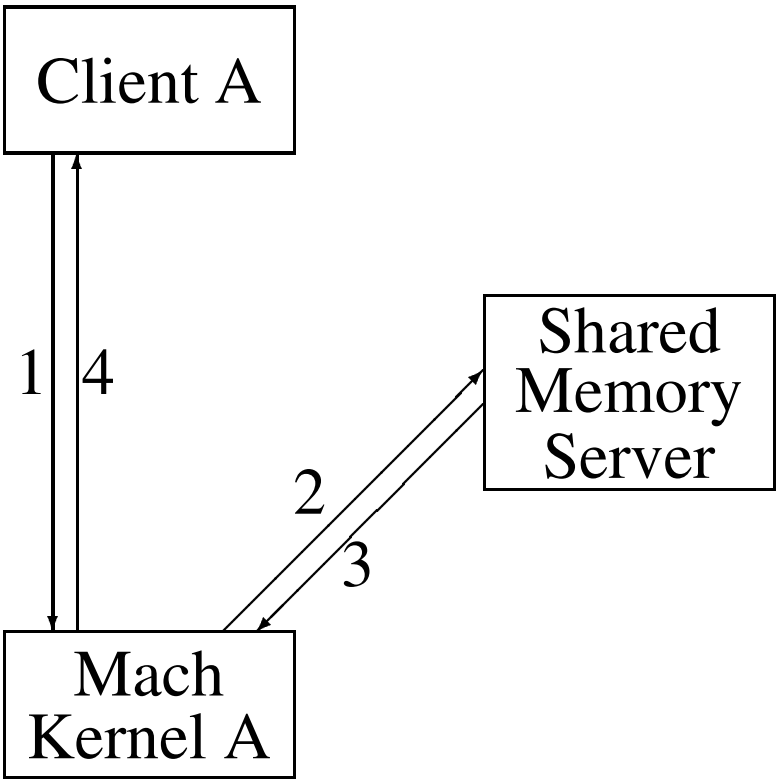
- Provide data page
- Restrict access to cached data
- Clean and/or flush cached data
- Allow data to be cached when not mapped
- Indicate that data is not available

# Example: Initialization



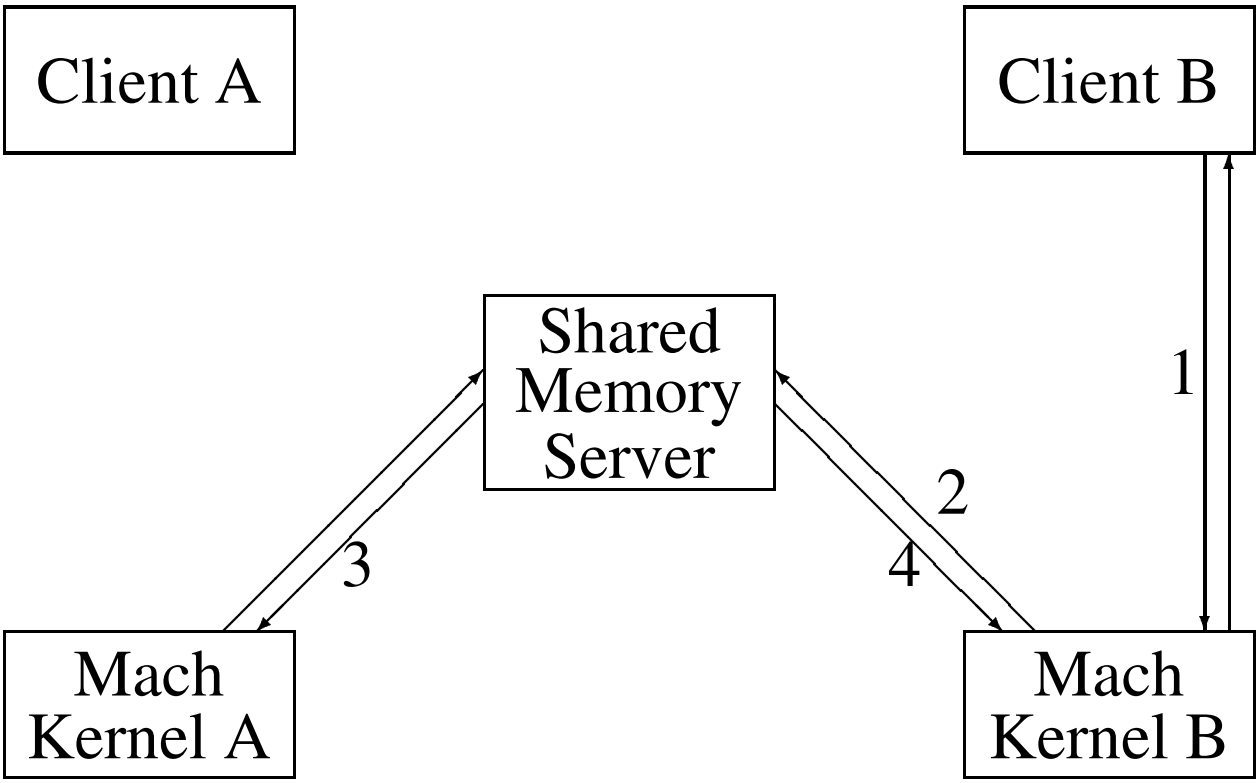

1. Request memory object from manager
2. Map object into address space
3. Kernel initializes memory object
4. Client resumed

# Example: Simple Fault



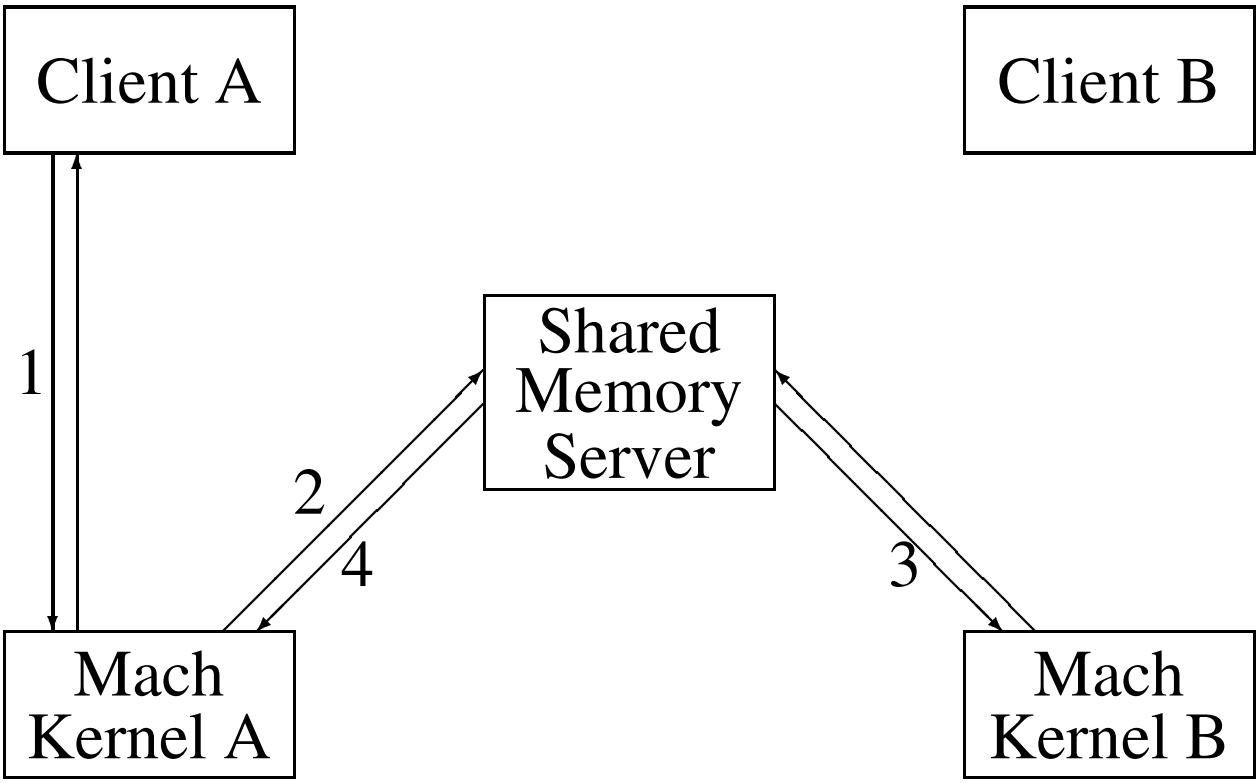

1. Client program page faults
2. Kernel requests data
3. Memory manager provides data
4. Kernel enters page mapping, resumes client

# Example: Shared Access




1. Another client read faults on same page
2. Its kernel requests data
3. Memory manager locks data to prevent modification
4. Memory manager provides data, read-only

# Example: Revoking Access




1. Client write faults on read-only page
2. Kernel requests write access to page
3. Memory manager requests flush
4. Write access granted after flush

# Potential Pitfalls

---

Memory managers cannot be completely trusted

- Failure to provide data
- Failure to free flushed data
- Spurious changes to data

Memory managers may use their own data

# Protecting User Programs

---

Obtain information about contents of memory

Multiple threads of control

- One thread handles page fault for another
- Access data of questionable origin in expendable thread

Analogy to communication failure

- Time out
- Notification

# Protecting the Kernel

---

Default Memory Manager

- Trusted provider of backing storage
- Used for pageout of zero-filled memory
- Backs data flushed to other managers

Copy-on-write

- Default memory manager backs original data if changed by manager
- Preserves original data

Wired-down memory – data will not disappear

Reserved memory pool for allocation of data structures needed to flush data