Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures

Matthias Lange

TU Berlin

June 1st, 2010
Background

- Richard Rashid, Avadis Tevanian, Micheal Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew
- presented at 2nd Symposium on Architectural Support for Programming Languages and Operating Systems, ACM October, 1987
Agenda

1 Introduction

2 Mach Design
   - Basic VM Operations
   - Memory Sharing

3 Implementation
   - Resident Memory
   - Address map
   - Memory Object
   - Shadow Objects
   - Pmap

4 Evaluation

5 Discussion
Introduction

- OS portability suffers from proliferation of memory structures
- Mach was a research project at CMU
  - considered a microkernel of 1st generation
  - developed as a portable multiprocessor OS
  - explore relationship between hardware and software memory architectures
**Task**  execution environment, basic unit of resource allocation, paged virtual address space

**Thread**  basic unit of CPU utilization

**Port**  communication channel, message queue protected by kernel, reference object

**Message**  typed collection of data

**Memory Object**  collection of data, can be mapped into an address space of a task
Basic VM Operations

- few assumptions about memory management hardware
  - handle and recover from page faults
- size of address space limited by hardware restrictions
- modification of address space through
  - allocate virtual memory
  - deallocate virtual memory
  - set protection status
  - specify inheritance properties
  - create and manage memory objects
Memory Sharing and Protection

- copy-on-write
- read/write
  - inheritance attribute shared, copy, none
  - specified on a per-page basis
- per-page current and maximum protection
  - combination of read, write, execute
  - enforcement depends on hardware support
Implementation

resident page table keep track of information about machine independent pages
address map entries describe mapping from addresses to a region of a memory object
memory object backing store managed by kernel or user task
pmap machine dependent mapping structure
Resident Memory

- Physical memory is treated as cache
- Information about physical pages maintained in page entries
- Page entry can be in
  - Memory object list
  - Memory allocation queues
  - Object/offset hash bucket
Address map

- address within a task are mapped to byte offsets in memory objects
- double linked list of address map entries
- contiguous area
- sorted ascending, no overlaps
- inheritance and protection attributes
- last fault hint
- typically small, u-area, code, stack, initialized and uninitialized data
### Memory Object

<table>
<thead>
<tr>
<th>ref counter</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>pager</td>
</tr>
<tr>
<td>paging name</td>
<td>paging object request</td>
</tr>
<tr>
<td>phys list</td>
<td>status info</td>
</tr>
</tbody>
</table>

- **reference counter** (allows for garbage collection)
- **pager** can use domain specific knowledge to retain non-referenced memory objects
- A managing task (pager associated)
Shadow Objects

- maintain information about pages of a memory object which have changed (CoW)
- create memory objects holding modified pages
- pointer to shadowed object, relies on the original object
- for read/write sharing sharing maps used
- address maps can point to memory objects or sharing maps
Pmap

- physical address map
- implemented in the machine dependent part
- ensures that the hardware map is operational on state transitions
Evaluation

- porting to new machines was relatively easy (few days to 4 weeks)
- Mach works on different platforms with different memory management hardware (IBM RT PC, VAX, SUN 3)
- VM operation performance better than UNIX
- on MP systems TLB consistency has to be manually ensured
Discussion

Questions

- Problems and implications on (soft) real-time workloads?
- Support for multiple page sizes (like 4KB and 4MB pages on IA32) at runtime?