Lecture 5: Virtual Memory Management in Mach

Mothy Roscoe and Joe Hellerstein

September 12, 2005

1 Background: Microkernels

What?

- Fashionable about 15 years ago: Mach, Chorus, Amoeba, etc.
- Key ideas: move functionality out of the kernel and into server processes (device drivers, paging, networking, Unix emulation, etc.)
- What's left? Varies: process scheduling, inter-process communication, maybe VMM hardware access.
- Example of what we saw before in SystemR with database catalogues: use the protection mechanisms that the OS provides to applications for implementing the OS itself.

Why?

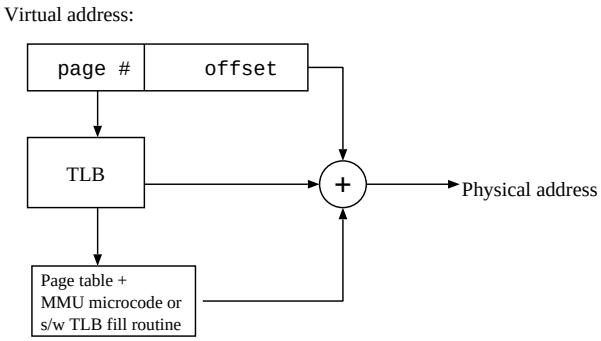
- Obvious benefit: robustness. If part of the OS misbehaves, it won't do bad things to the rest of the OS.
- Wasn't actually a problem in commercial operating systems at the time, since # kernel programmers was very small. Big problem these days for Windows and Linux, but neither are μKernel based. More in Steve Hand's lecture later this month.
- Further benefit: user-processes can perform operating system functionality, such as paging. Why on earth would you want this? Well, to change the *policy* used or employ multiple policies, rather than the one the kernel imposes.

In fact, Mach 3.0 was technically a μKernel, but prior versions were not since the BSD server was also in the kernel. Mac OS X is based on Mach 3.0, but they've also put pretty much everything back in the kernel.

Aside: separation of "mechanism" and "policy": mechanism is the kind of operation you do and how you actually achieve it (which can have protection applied), while policy of precisely how you want to use the mechanism at a particular point. Mach championed the idea of separation of mechanism and policy, idea was that mechanism (or that bit which needed to be privileged) was in the kernel, but policy was in user space.

- Critical challenge for μKernels: Inter-Process Communication (IPC) performance.
- Idea: use the virtual memory system:
  - data sharing between processes *(tasks)*
  - bulk data transfer
  - distributed communication,
  - etc.

2 Review of Virtual Memory



Note that the page table can be hardware maintained (e.g. on VAX, ia32 or mc68k) or software (MIPS, Alpha). The format can therefore be defined by the OS or by the hardware, though in practice sometimes the TLB fill routine for a software-based MMU is so performance critical that only one format is really possible (e.g. MIPS).

3 Mach

Aims:

- wide utility and flexibility (in particular, IPC)
- portability (hardware independence)
- multiprocessor support (but actually not much really)

API design (not user-level pagers):

- How to represent VM to user programs
- Separates *virtual address space* management from *physical memory management*
- Virtual memory allows allocation / deallocation / setting protection on a per-page basis.
- Physical memory represented by *memory objects* (another level of indirection).
- Memory objects are implemented by *pagers*, which can fill page faults any way they choose.
- Subranges of memory objects mapped into regions of virtual address space

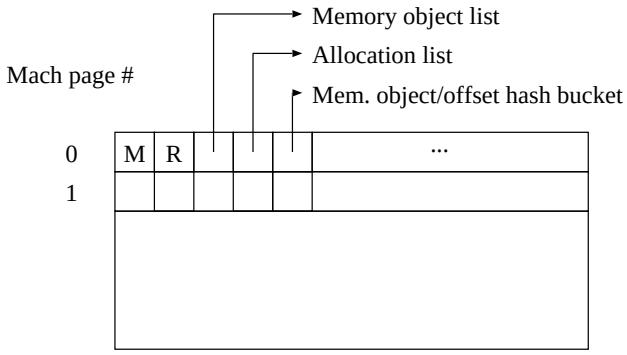
Aside: Note that a virtual address space can be manipulated by anything that can acquire a reference to the task, not just the process itself. This is a (simplified) instance of *capability-based* protection, which we'll cover later (possibly next semester).

Question: Why is the virtual address space manipulated on a per-(Mach)-page basis rather than representing regions themselves as objects (so they look like segments)?

Implementation:

- Resident page table: machine-independent page structures
- Address map: representation of an address space
- Memory objects (see above)
- The *pmap*: the machine-dependent bit!

The Resident Page Table:



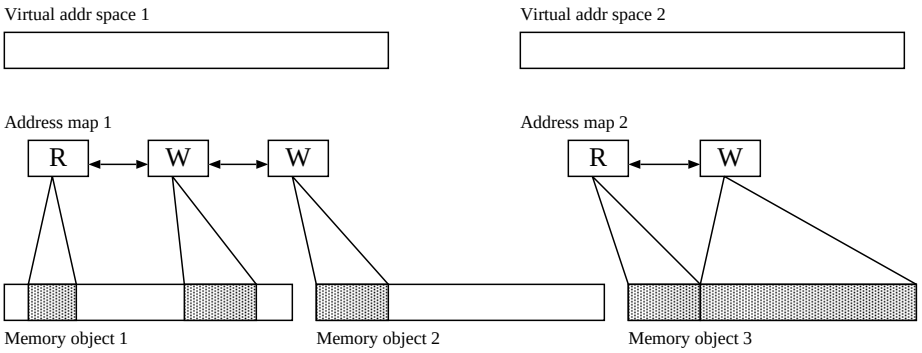
Basically, an *inverted page table*.

The PMap:

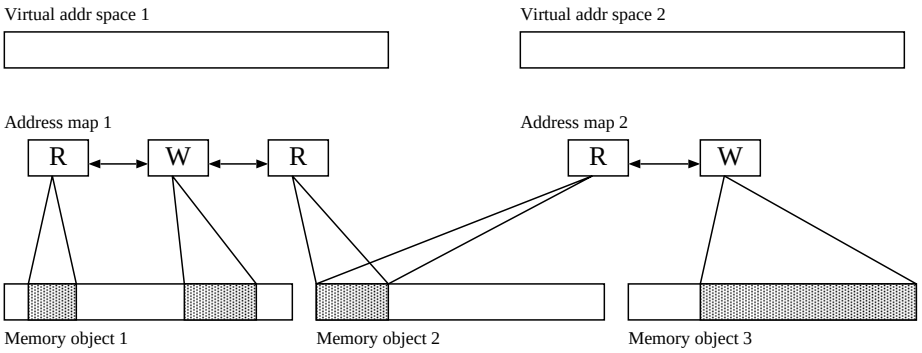
- Purely “soft-state”: refreshed at any time from the RPT - nice idea!
- Casually described as `pmap.c`, but could be very large for a source file (10,000’s of lines).
- Glossed over: how does the pmap return dirty/modified bits back to the RPT?

Address maps and copy-on-write

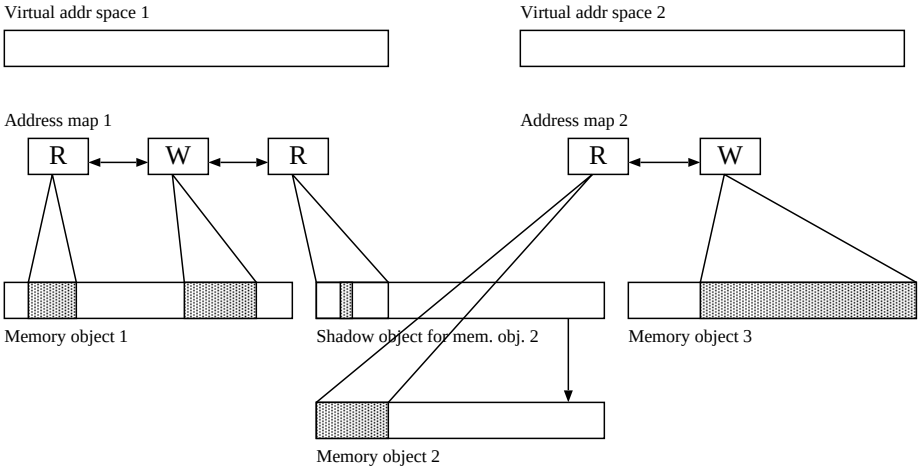
How to achieve efficient VM copy operations (both intra- and inter-address space). Basic use:



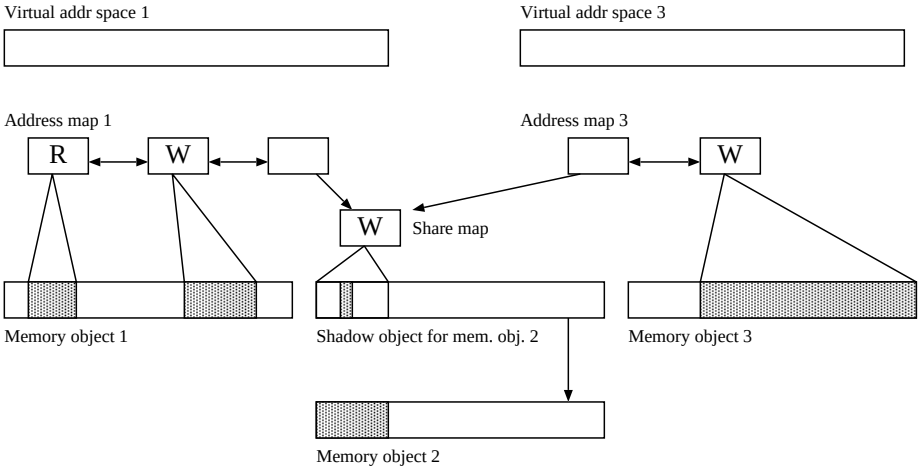
Copy-on-write (without modification):



Copy-on-write (with modification): shadow objects



Read-write sharing with share maps:



4 Mach summary

Key points:

- Machine-independent, portable VM design.
- Multi-address-space sharing of memory objects through address maps, share maps, shadow objects.
- User-definable pagers support many usage models of VM hardware

Key flaws:

- Very little discussion of MP issues (such as TLB shootdown)
- Many issues glossed over
- Performance evaluation is weak
  - And indeed, as Appel and Li show, it’s unimpressive.

– Mach `fork` vs. UNIX `fork` is an unfair comparison: `vfork` is a better comparison.

– Larger page sizes (Mach pages vs. h/w pages) lead to much of the performance improvement, rather than the design proper.

5 Virtual Memory Primitives for User Programs

Basic idea: virtual memory hardware provides efficient support for (1) translation, and (2) traps on page faults. Use the traps to detect arbitrary memory references with very low overhead, and use this for:

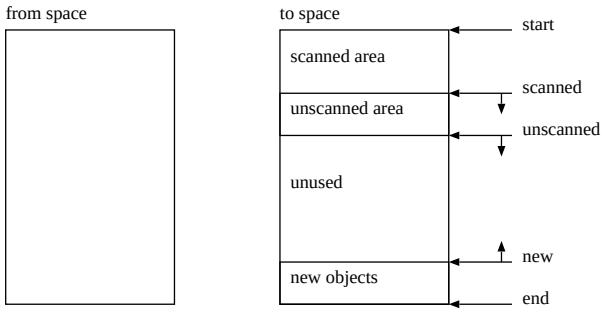
- Concurrent garbage collection
- Distributed shared memory
- Concurrent checkpointing
- Persistent programming
- Extended addressing
- Paging to main memory with compression
- Heap overflow detection

Performance implications

- Once you start using the VM hardware for things other than paging to disk (which is what it was designed for), a different set of characteristics (for both hardware and the OS) becomes important.
- Trap dispatch performance is now important (it's not when it's completely dominated by disk seek time).
- Previously little-used functionality (in particular, MAP2) becomes a lot more useful: allowing two threads to access the same memory with different permissions by mapping it in two different places.

Concurrent Garbage Collection

Described quite briefly in the paper.



Invariants maintained:

- Mutator thread sees only to-space refs (stack and register refs copied at start of collection cycle)
- Newly-allocated objects only contain to-space references
- Scanned-area objects only contain to-space references
- Unscanned-area objects contain both to-space and from-space references

How is the VM hardware used?

- From-space is marked no-access to trap all mutator references to from-space (and initiate copying)
- Unscanned area is marked no-access to trap all mutator references to unscanned (but copied) objects.
- Also provides synchronization between collector and mutator
- Need MAP2 facilities so that collector thread can access protected pages in unscanned are and from-space

Turns out that "emulating" MAP2 with PROTN, TRAP and UNPROT without too much loss in efficiency - artifact of typical access patterns.

6 Apple and Li Summary

Key points:

- VM facilities have uses other than large addressable memories
- Key hardware feature is detecting memory references and calling application $\Rightarrow$ smaller page sizes are better.
- OS should provide fast implementations of page faulting and protection changes
- Example of different ways of using OS abstractions: "Most good systems research is about the creative abuse of technology".

Key flaws:

- Small pages are not necessarily a good thing. Not much discussion of other tradeoffs.