Lecture 6: Lightweight Recoverable Virtual Memory, and Virtual Memory Primitives for User Programs

Mothy Roscoe and Joe Hellerstein

15 September 2005

1 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

What do you need?

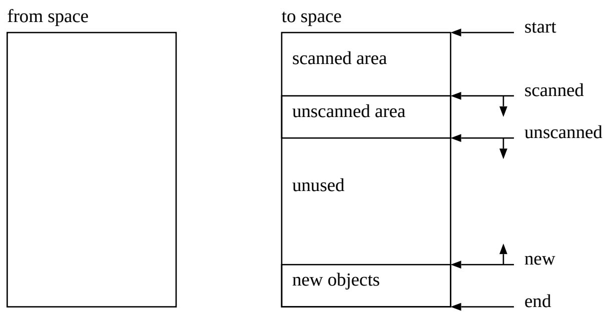
- TRAP handle page-fault traps in user mode. Note that all you mostly need is to handle protection faults and restart accesses, rather than actually do paging (avoiding liability inversion issues with page reclamation).
- PROT1 decrease accessibility of a page: the kernel shouldn't really object to this
- PROTN: decrease accessibility of several pages: largely a performance optimization, but an important one.
- UNPROT: increase accessibility of a page: clearly this requires an OS check as to "real" page accessibility. Authors show that UNPROTN is never really needed.
- DIRTY: return list of pages dirtied (not accessed, but dirtied) since the previous call. Note that the application could (inefficiently) do this itself using PROT, UNPROT, and TRAP.
- MAP2: map the same physical page at two different places, with different protection rights. Turns out to be the most tricky one (because of some hardware), and hard to emulate.

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.

Performance:

Why is Mach so slow? Speculation:

- Unix emulation (they don't use native Mach facilities)
- Mach is simply unoptimized
- Overhead of Mach's portability

TLB shootdown overhead is not so bad, since memory is made less accessible in large batches (one shootdown) and made more accessible in single pages (can be patched up after the fact).

2 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.

3 Lightweight Recoverable Virtual Memory

Goal: allow Unix applications to manipulate:

- in-memory data structures
- transparently persisted to disk
- well-defined failure semantics (committed data)

Particular application in mind: metadata management for the Coda file system server process.

Consistency and recovery from crashes is a central problem in file system design and implementation, as we’ll see later.

Team had previously used Camelot: full distributed transaction facility built by Spector et. al.

- Overhead of multiple address spaces, context switching, IPC, etc. Probably outweighed by network communication in distributed case, hence not a problem for Camelot.
- Heavyweight facilities impose additional constraints on programmers (*though argument here is not well made*).
- Portability limited by Camelot’s reliance on Mach (in particular external paging), and difficulty tracing bugs in combinations of research systems.

Q. Where is this in the design space for transactions? A. Good question …illustrates of different perspective of databases systems and operating systems.

Architecture:

- Very lightweight - not a lot of magic going on, just explicit modifications and write-ahead logging
- External “data segments” on persistent storage
- Processes “map” (or, “read” as we sometimes say) regions of data segments into their address space (c.f. Mach).
- No aliasing - since system needs to know when things are modified, and …
- transactions are initiated by “set_range’: copies old values so they can be efficiently restored (in memory) after an abort.

What’s left out:

- Distributed transactions
- Nested transactions
- Media resiliency

Options: having thrown out functionality from the system, further options reduce the semantics still further:

- *no-flush* means committed transactions are not written to the log until an explicit call is made (allows batching). Where might you use this?
- *no-restore*: user commits (as it were) to never aborting the transaction, hence no need for undo except in crash recovery.
- Very systems-ish: not popular with database folks. Lots of stuff exposed.

Implementation:

- Log only contains new values, since uncommitted changes never written to disk (NO-STEAL).
- "Decoupled from the VM system" ⇒ in-memory representation is paged independently of data segment
- Log truncation: at time of writing, "stop the world" technique (reuse the crash recovery functionality). Incremental truncation is subtle and untested ("in process of debugging").

Optimizations:

- Intra-transaction: coalesce set_range calls, useful (even essential) since programmers use them willy-nilly.
- Inter-transaction: coalesce no-flush log records when you actually do the flush.

Key points:

- Goal: lightweight resusable facility to allow programs to manipulate persistent data structures with well-defined failure semantics
- Experience with fully general transaction system convinced the team that they wanted something less general, more lightweight that only provides recoverable memory.
- Result considerably outperformed Camelot across the board

Question: Why aren't general TP managers constructed in this way, and allow access to subsets of functionality? (c.f. top-down vs. bottom-up systems approaches). Of course, the paper doesn't demonstrate this …

Possible criticisms:

- Many of the objections to Camelot refer to heavyweight engineering techniques, rather (arguably) than system design. Unclear that it's RVM's innovations (simplicity and open design) that address the deficiencies of Camelot for this application.
- Performance is better, but the authors gloss over the use of `set_range`. This eliminates any memory trap overhead by putting the burden on the programmer. Clearly could be eliminated by use of Appel & Li's PROT and TRAP.
- *Persistent errors:* errors in a set range region aren't fixable except by hand.