Virtual Memory Primitives for User Programs

Andrew Appel and Kai Li

Objectives

- User programs can benefit from use of VM primitives
  - Efficiency is important, not dominated by disk access overhead
  - Many examples
- What's the performance on set of OS/arch
- Some design considerations

VM Primitives

- Trap: Handle page-fault traps in user mode
- Prot1: Decrease accessibility of a page
- ProtN: Decrease accessibility of N pages
  - More efficient than calling Prot1 N times
- Unprot: Increase accessibility of a page
- Dirty: Return list of written pages since last call
  - Can be emulated with ProtN, Trap, and Unprot
- Map2: Map same physical page at two different virtual addresses, at different access levels, in the same address space

Example: Concurrent Garbage Collection

- From-space and to-space
- Traverse reachable objects and move from from-space to to-space; eventually discard from-space
- ProtN from-space and initiate gc (collector)
- Do not block running threads (mutator)
- On mutator Trap – move object and Unprot
- Must make sure in same process collector and mutator have different privileges – Map2
- Don’t have to worry about synchronization – taken care by protection restriction
- Benefits from small page size

Example: Virtual Shared Memory

- One writer multiple readers
- Trap, Prot1, Unprot – to modify read/write access to shared pages; to get current copy of remote page
- Map2 – trap handler needs access to page protected from clients
- Small page size useful (for false sharing)

Example: Concurrent Checkpointing

- Simple approach: stop all threads, copy all state, restart all threads
- More efficient approach:
  - ProtN state and start copying/checkpointing
  - On Trap – copy and Unprot
  - For repeated checkpoints: Dirty works best
  - Suggestion: medium page sizes (why?)

Other examples:

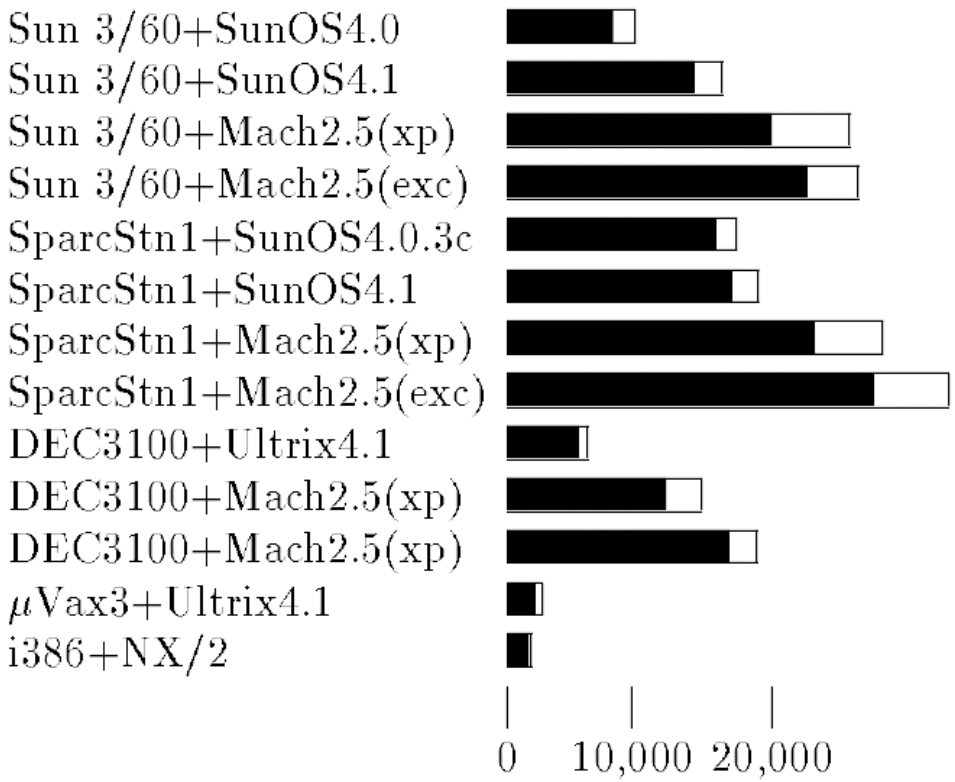
- Generational garbage collection
- Persistent stores
- Extending addressability
- Data-compression paging
- Heap overflow detection

Methods	TRAP	PROT1	PROTN	UNPROT	MAP2	DIRTY	PAGESIZE
Concurrent GC	√		√	√	√		√
SVM	√	√		√	√		√
Concurrent checkpoint	√		√	√		‡	√
Generational GC	√		√	√		‡	√
Persistent store	√	√		√	√		
Extending addressability	√	*	*	√	√		√
Data-compression paging	√	*	*	√	√		
Heap overflow	√		†				

How did real systems perform

Machine	OS	ADD	TRAP	TRAP +PROT1 +UNPROT	TRAP +PROTN +UNPROT	MAP2	PAGESIZE
Sun 3/60	SunOS 4.0	0.12	760	1238	1016	yes	8192
Sun 3/60	SunOS 4.1	0.12		2080	1800	yes	8192
Sun 3/60	Mach 2.5(xp)	0.12		3300	2540	yes	8192
Sun 3/60	Mach 2.5(exc)	0.12		3380	2880	yes	8192
SparcStn 1	SunOS 4.0.3c	0.05		*919	*839	yes	4096
SparcStn 1	SunOS 4.1	0.05	†230	1008	909	yes	4096
SparcStn 1	Mach 2.5(xp)	0.05		1550	1230	yes	4096
SparcStn 1	Mach 2.5(exc)	0.05		1770	1470	yes	4096
DEC 3100	Ultrix 4.1	0.062	210	393	344	no	4096
DEC 3100	Mach 2.5 (xp)	0.062		937	766	no	4096
DEC 3100	Mach 2.5 (exc)	0.062		1203	1063	no	4096
μVax 3	Ultrix 2.3	0.21	314	612	486	no	1024
i386 on iPSC/2	NX/2	0.15	172	302	252	yes	4096

Or scaled performance…



- Numbers suggest it’s possible to get good performance, but not necessarily what’s done…

Design Considerations

- Make pages more accessible one at a time, less accessible in batches (ProtN)
  - TLB flushes on multi-processors
- Page size – small?
- Map2:
  - Alternatives: system call to copy to-from protected area (3x), put service in different process, put service in kernel -> not good…
  - Map2 with physically addressed caches – ok
  - Virtually addressed: consistency challenge
- Pipelining:
  - Can you undo instruction effects after page fault
  - Most of the examples are semi-synchronous, so ok.