64-bit address spaces

- **Straight hierarchical page tables not efficient**
- **Solution 1: Guarded page tables [Liedtke]**
  - Omit intermediary tables with only one entry
  - Add predicate in high level tables, stating the only virtual address range mapped underneath + # bits to skip
- **Solution 2: Hashed page tables**
  - Store Virtual $\rightarrow$ Physical translations in hash table
  - Table size proportional to physical memory
  - Clustering makes this more efficient
OS policy choices (besides page table)

- **Page replacement**
  - Optimal – Least soon to be used (impossible)
  - Least recently used (hard to implement)
  - Random
  - Not recently used

- **Direct-mapped physical caches**
  - Virtual $\rightarrow$ Physical mapping can affect performance
  - Applications can conflict with each other or themselves
  - Scientific applications benefit if consecutive virtual pages to not conflict in the cache
  - Many other applications do better with random mapping
4.4 BSD VM system

- Each process has a *vmspace* structure containing
  - *vm_map* – machine-independent virtual address space
  - *vm_pmap* – machine-dependent data structures
  - statistics – e.g. for syscalls like *getrusage()*

- *vm_map* is a linked list of *vm_map_entry* structs
  - *vm_map_entry* covers contiguous virtual memory
  - points to *vm_object* struct

- *vm_object* is source of data
  - e.g. vnode object for memory mapped file
  - points to list of *vm_page* structs (one per mapped page)
  - *shadow objects* point to other objects for copy on write
Pmap (machine-dependent) layer

- Pmap layer holds architecture-specific VM code
- VM layer invokes pmap layer
  - On page faults to install mappings
  - To protect or unmap pages
  - To ask for dirty/accessed bits
- Pmap layer is lazy and can discard mappings
  - No need to notify VM layer
  - Process will fault and VM layer must reinstall mapping
- Pmap handles restrictions imposed by cache
Example uses

- *vm_map_entry* structs for a process
  - r/o text segment → file object
  - r/w data segment → shadow object → file object
  - r/w stack → anonymous object

- **New *vm_map_entry* objects after a fork:**
  - Share text segment directly (read-only)
  - Share data through two new shadow objects (must share pre-fork but not post fork changes)
  - Share stack through two new shadow objects

- **Must discard/collapse superfluous shadows**
  - E.g., when child process exits
What happens on a fault?

- Traverse `vm_map_entry` list to get appropriate entry
  - No entry? Protection violation? Send process a SIGSEGV

- Traverse list of [shadow] objects

- For each object, traverse `vm_page` structs

- Found a `vm_page` for this object?
  - If first `vm_object` in chain, map page
  - If read fault, install page read only
  - Else if write fault, install copy of page

- Else get page from object
  - Page in from file, zero-fill new page, etc.
DEC Alpha MMU

- **Software managed TLB (like MIPS)**
  - In model used by paper: 8K base pages,
    TLB supports 128 instruction/128 data entries

- **But TLB miss handler not part of OS**
  - Processor ships with special “PAL code” in ROM
  - Processor-specific, but provides uniform interface to OS
  - Paper calls this firmware, but runs from memory like OS

- **Various events vector directly to PAL code**
  - CALL_PAL instruction, TLB miss/fault, FP disabled

- **PAL code runs in special privileged processor mode**
  - Interrupts always disabled
  - Have access to special instructions and registers
PAL code interface details

- Examples of Digital Unix PALcode entry functions
  - callsys/retsys - make, return from system call
  - swpctx - change address spaces
  - wrvptptr - write virtual page table pointer
  - tbi - TBL invalidate

- Some fields in PALcode page table entries
  - GH - 2-bit granularity hint → $2^N$ pages have same translation
  - ASM - address space match → mapping applies in all processes