Complete Virtual Memory Systems

- two examples
  - VAX/VMS – 1970s/early 1980s
    - the origins of many strategies still in use
  - Linux
    - current OS that runs on a wide variety of sizes of systems

VAX/VMS

- system architecture: VAX-11
  - wide range of systems (low-end to high-end)
- OS: VAX/VMS (or VMS)

VAX 11/780
introduced 1978
dimensions: 60” x 48” x 30”
originally supported up to 8MB of memory
29-bit physical address space (512MB max)
cost: $120,000-160,000

https://www.computerhistory.org/revolution/mainframe-computers/7/182/736

VAX/VMS

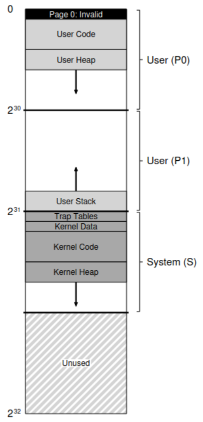
- memory management hardware
  - 32-bit virtual address space per process
  - page size 512 bytes
  - hybrid paging and segmentation
    - upper 2 bits of VPN designated the segment
  - base and bounds registers – for each segment
    - base holds the address of the page table for the segment
    - bounds holds the size (number of PTEs)

23-bit VPN,
9-bit offset

https://www.computerhistory.org/revolution/mainframe-computers/7/182/736

VMS Address Space

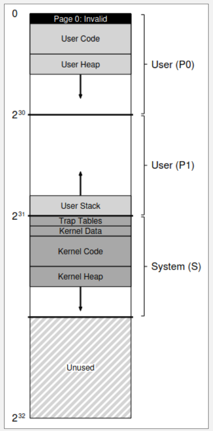
- lower half of the address space used as “process space”
  - unique to each process
  - first half of process space (P0) used for the program code and heap
  - second half of process space (P1) used for the stack
- upper half of the address space used as “system space”
  - holds protected OS code and data
  - shared across processes
    - S base, bound registers are not updated on context switches
  - allows kernel to appear as a protected library to applications
    - doesn’t require special handling for copy data for system calls or swapping
- PTE contains a protection bits specifying who can access (user or kernel)



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-complete.pdf

VMS Address Space

- page 0 is always invalid
  - helps detect null-pointer accesses
    - PTE for 0x0 has the invalid bit set, which generates a segmentation fault



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-complete.pdf

VAX/VMS

- dealing with the very small page size (512 bytes)
  - separate page tables for P0 and P1 regions
  - user page tables placed in kernel VM
    - kernel can swap page table pages out if needed
- address translation
  - hardware-managed TLB
  - on TLB miss, hardware must look up system page table (in physical memory) to find the address of the page in the page table, then look up the actual page table entry

VAX/VMS

- PTE contains
  - valid bit
  - protection field (4 bits)
  - dirty bit
  - field reserved for OS use (5 bits)
  - PFN
  - (no reference bit)
- page replacement
  - no reference bit
    - can emulate via protection and reserved OS bits
  - global page replacement schemes like LRU are susceptible to *memory hogs*

VAX/VMS

- *segmented FIFO* replacement policy
  - each process has a *resident set size* (RSS) specifying the maximum number of pages it can have in memory
  - when a process P_1 exceeds its RSS, the first-in of its pages is moved to a (global) *second-chance list*
    - *clean-page free list* – for clean (unmodified) pages
    - *dirty-page list* – for dirty (modified) pages
  - when a process P_2 needs a free page, it takes the first free page from the clean-page list
  - if P_1 faults on a page still on a second-chance list, it is reclaimed
    - avoids disk access
  - with sufficiently big second-chance lists, performance is close to LRU
- second-chance lists also allow for clustering
  - dirty pages written in batches for more efficient disk I/O
  - moved to clean list after writing

VAX/VMS

- *demand zeroing*
  - instead of zeroing a page when it is allocated, a page table entry is added but marked “inaccessible”
    - uses the “reserved for OS” bits to tag as a demand-zero page
  - when the page is accessed, the trap handler finds a new page, zeros it, and maps into the processes address space

    → saves the work of zeroing for a page that is never accessed

- *copy-on-write*
  - when a page is copied from one address space to another, just map the page into the other address space and mark “read only”
    - only copy when/if the page is written to
    - beneficial with `fork()`/`exec()`