Complications of Multiprogramming

• Makes it hard to allocate space contiguously
  - Convenient for stack, large data structures, etc.

• Need fault isolation between processes
  - Someone else testing tcpproxy on your machine…

• Processes can consume more than available memory
  - Dormant processes (waiting for event) still have core images
Solution: Address Spaces

• Give each program its own address space

• Only “privileged” software can manipulate mappings

• Isolation is natural
  - Can’t even name other proc’s memory
Alternatives

• **Segmentation**
  - Part of each memory reference implicit in segment register
    \[
    \text{segreg} \leftarrow \langle \text{offset}, \text{limit} \rangle
    \]
  - By loading segment register code can be relocated
  - Can enforce protection by restricting segment register loads

• **Language-level protection (Java)**
  - Single address space for different modules
  - Language enforces isolation
Paging

- Divide memory up into small “pages”
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “dirty” bit
  - Control caching of page
Example: Paging on PDP-11

- 64K virtual memory, 8K pages
- 8 Instruction page translations, 8 Data page translations
- Swap 16 machine registers on each context switch
Example: VAX

- **Virtual memory partitioned**
  - First 2 Gigs for applications
  - Last 2 Gigs for OS—mapped same in all address spaces
  - One page table for system memory, one for each process

- **Each user page table is 8 Megabytes**
  - 512-byte pages, 4 bytes/translation,
    1 Gig for application (not counting stack)

- **User page tables stored in paged kernel memory**
  - No need for 8 physical Megs/proc. only virtual
Example: MIPS

- Hardware has 64-entry TLB
  - References to addresses not in TLB trap to kernel

- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global

- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses

- User TLB fault handle very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables
Example: Paging on x86

- Page table: 1024 32-bit translations for 4 Megs of Virtual mem
- Page directory: 1024 pointers to page tables
- %cr3—page table base register
- %cr0—bits enable protection and paging
- INVLPG – tell hardware page table modified
OS effects on application performance

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

- Direct-mapped physical caches

- Page table structures
  - Left to OS on architectures like MIPS
  - Hashed vs. hierarchical page table affects performance
Paging in day-to-day use

- Demand paging, demand zero-fill
- Shared libraries
- Shared memory
- Copy-on-write (fork, mmap, etc.)
- Optimized unix pipes
Benefits and disadvantages

• What happens to user/kernel crossings?
  - More crossings into kernel
  - Pointers in syscall arguments must be checked

• What happens to IPC?
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on x86
Example: 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.