Virtual Memory

Dongyoon Lee

Without Virtual Memory

- Program address = RAM address
- Program address space
  - x86_32 (32-bit): 4GB
  - x86_64 (48-bit): 256TB
- Physical RAM memory
  - Dynamic Random Access Memory (DRAM)
  - Limited in size: e.g., 16GB
- A program crashes if we try to acess more RAM than we have.

Virtual Memory

- Permits a process to be run without having the entire contents of its address space loaded into main memory at one time.
- Program address maps to RAM address.
  - The part of the address space loaded into main memory is called resident.
  - The remainder is called nonresident. Nonresident data is saved on a secondary storage area, called the backing store.

What is Virtual Memory Good for?

- VM increases the degree of multiprogramming.
  - More runnable processes can be kept in main memory at one time
  - More runnable processes means increased CPU utilization
- VM allows running large applications whose address space exceeds the amount of main memory.
- Memory-mapped files provide an useful alternative to traditional I/O system calls.

Page Tables

- Virtual address space is partitioned into fixed-size pages (e.g., 4KB).
- Physical memory is partitioned into the same size page frames.
- A Page Table define a mapping from virtual pages to physical pages.
  - Per process
  - Resides in memory
- Page Table Entry (PTE)
  - Map VPN (virtual page number) to PPN (physical page number)
  - Flags: present, RWX permission, used, dirty, etc.

Page Tables

Size of Page Tables

- On real hardware, a full set of page tables for a process can be very large.
- On x86_32
  - each PTE consumes 4 bytes, and maps one 4K page.
  - 32-bit 4GB address spaces = 1M PTEs = 1M * 4B = 4MB
- Each process needs its own individual page table.

Multi-level Page Tables (Radix Tree, Trie)

- x86_32: a 2-level page table
  - The first 10 bits of virtual address are used as an index into a 4KB Page Directory.
  - The next 10 bits of virtual address are used as an index into a 4KB Page Table, which need not be resident.
  - Each PTE maps a 4KB page, which need not be resident.
- x86_64: a 4-level page table

x86_32 Segmentation and 2-level Paging

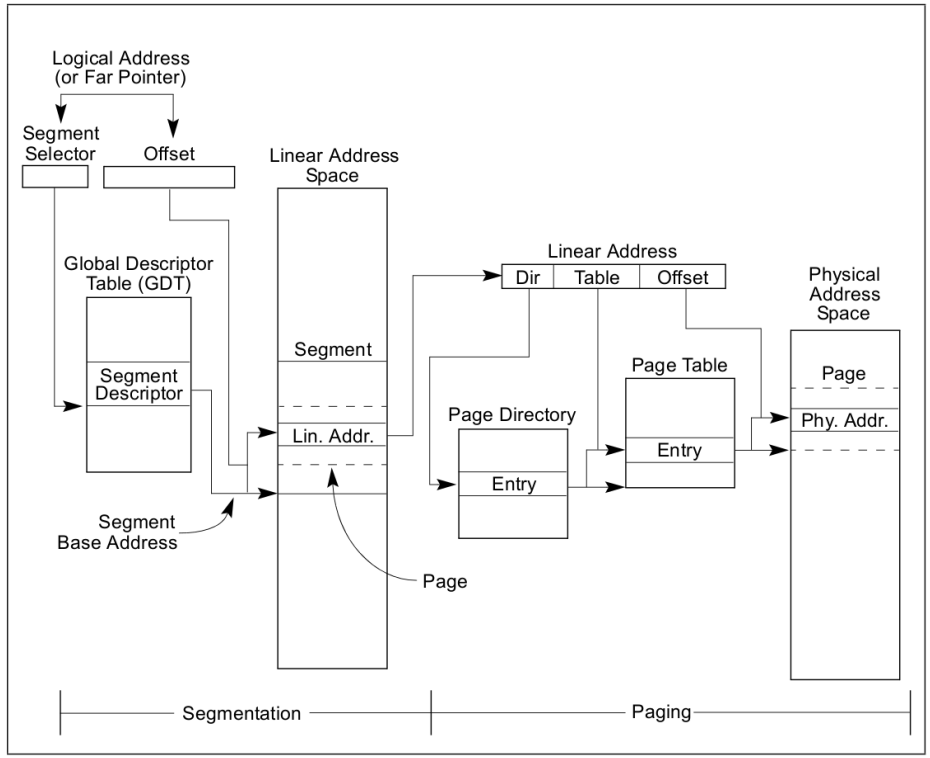
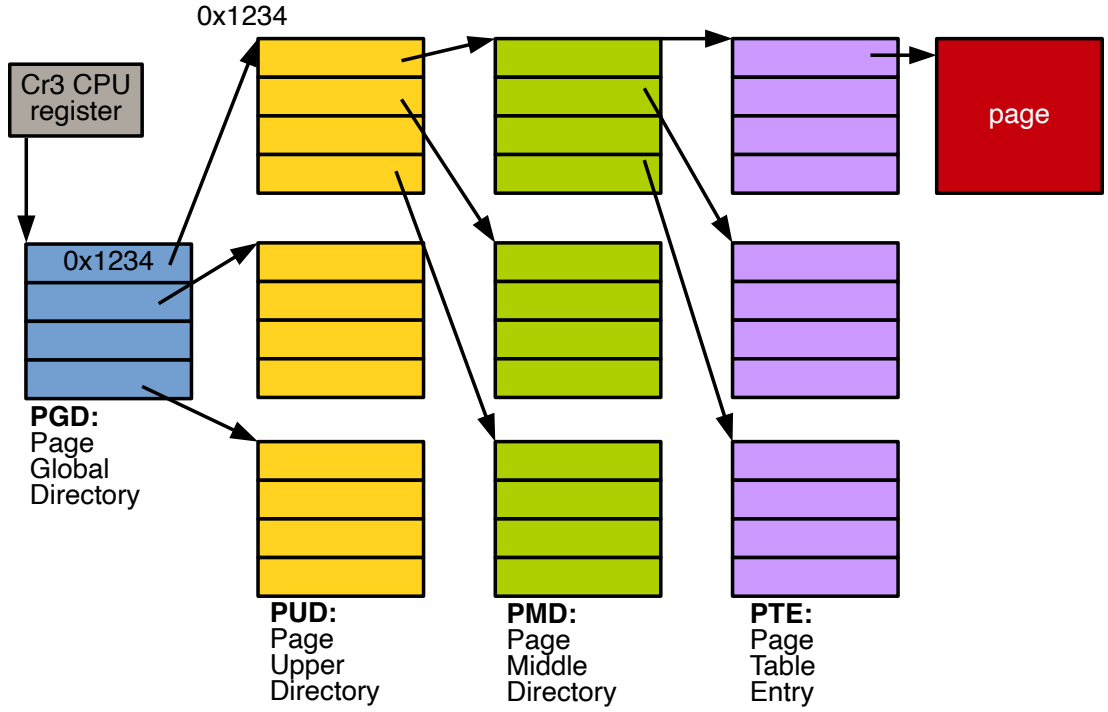


Figure 3-1. Segmentation and Paging

x86_64 4-level Paging



Hardware Support: MMU and TLB

- Virtual memory requires hardware support to work efficiently.
- Address translation is performed by the Memory Management Unit (MMU), a hardware unit, on every memory reference.
- To avoid repeated access to page tables in main memory, a hardware cache called the Translation Lookaside Buffer (TLB) is used to store recently used PTE's.

How does it work?

- Nonresident pages are marked invalid in the page table.
- Attempts to access invalid pages are detected by MMU, causing a page fault trap to the OS.
- The page fault handler “fixes things up”:
    1. Allocating a page of memory.
    2. Retrieving data from backing store.
    3. Marking page table entry valid.
    4. Allowing applicaiton to continue.
- VM is transparent to user applications.

Performance Issues

- Accesses to resident pages execute at main memory speed.
- Accesses to nonresident pages require I/O, and are therfore, slower.
- The effective execution rate of a process under virtual memory depends on the fraction of accesses to nonresident pages.

Locality of Reference

- Typically, there are fairly long time intervals during which a process acecsses only a fraction of its address space.
- Having only 1/2 or 1/3 of pages reisdent is often sufficient to achieve extremely low page fault probability.

Hierarchies of Storage (Caching)

- Virtual memory is one example of an important principle that reappears throughout hardware and software design:
- Whenever there is significant locality of reference, overall system performance can be improved by using hierarchies of storage.
- The performance improvement is achieved by trading small, fast, but expensive memory for large, slow, but cheap memory.

Caching: The idea

- Caching is based on the following ideas:
  - Keep data in active use in small, fast, expensive memory.
  - Keep inactive data in large, slow, but cheap memory.
  - Transfer data to fast memory on demand.
- The idea extends to multiple levels of memory hierarchy: (e.g. registers, L1 cache, L2 cache, RAM, disk, tape).

Page Fault Handling: MMU (HW)

- A page fault exception occurs when a process attempts to access a virtual page number for which the corresponding page table entry is marked invalid.
- Access is trapped by the MMU, and control goes to the kernel, as if a system call or other exception had occurred.
- The kernel must now service the page fault.

Page Fault Handling: OS

1. Locate the data on backing store.
2. Get a free page of main memory.
3. Perform I/O to bring data into main memory.
4. Install valid mapping in page table.
5. Return to user mode to retry faulting instruction.

Page Replacement

- The VM subsystem must perform page replacement to compensate for pages consumed by page faults.
- A victim page is selected according to a replacement policy.
- All mappings of the victim page are invalidated.
- Contents of the page are written to backing store.
- The page is added to the free memory pool.

Page Replacement Policies

- First-in, First-out (FIFO)
  - Fail to take locality into account
- Optimum (OPT)
  - Replace the page that will not be used for the longest time in the future
  - Require predicting the future. Used for theoretic comparison.
- Least-Recently Used (LRU)
  - Replace the page that has not been accessed for the longest time.
  - Assumes that past accesses are a good predictor of future.

The ``Use'' Bit

- Many hardware MMUs provide a use bit, which is a bit in each PTE that is set when the PTE is used in translation.
- Use bits can be used to approximate LRU.
- Periodically (via timer interrupts), check and clear use bits and update summary information (e.g. "last use" timestamp or "access count").
- Use the summary information as a basis for selecting victim pages.

The ``Dirty'' Bit

- Many MMUs also provide a dirty bit, which is set each time a write access is performed using the PTE.
- If a page has been written since being loaded from backing store, it is dirty otherwise it is clean.
- Clean pages don't have to be written to backing store if they are selected as victims.
- Selecting clean pages as victims saves half the I/O.

Memory-Mapped Files

- A memory-mapped file is a file that has been made to appear as though it has been loaded into memory.
- Access to the mapped region causes page fault, which causes data to be loaded from the backing file.
- Writing to the mapped region marks pages dirty, and arranges for them to be written back to the file.

The mmap() System Call

- Modern Unix/Linux systems support the mmap() system call, for selectively mapping and unmapping regions of virtual address space.
- void *mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset);
  - addr: specified VA or allows system to choose
  - length: length of region
  - prot: PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC
  - flags: MAP_SHARED, MAP_PRIVATE, possibly with MAP_ANONYMOUS
  - fd: the file descriptor of a backing file (unless MAP_ANONYMOUS)
  - offset: the starting offset within the backing file.

Sharing of Pages

- OS may permit sharing of pages between multiple address spaces.
  - Several processes might share a single read-only copy of executable code.
  - Processes might use shared read-write pages to communicate.
- When a page is chosen for eviction, mappings must be invalidated in page tables for all processes sharing that page.

Copy-on-Write (CoW)

- Processes can share writeable data (and code!), up until the first time it is written.
- Shared pages initially set to read-only in page table.
- On page fault due to write access, a private copy is made and set to read/write in page table.
- Subsequent accesses by writer are to private copy; others can continue to share read-only copy.
- Important for dynamically loaded libraries, because these often contain read/write variables.

Efficient fork()/exec() via CoW

- On fork(), there is no reason to initially copy all code and data for the parent process.
- Instead, only the page tables are copied, and the shared pages are set to read-only.
- When either parent or child attempts to write, a page fault occurs and a private copy is made.
- Pages that are not written do not have to be copied at all.
- Can even use COW on the page tables themselves! (saves overhead for fork() immediately followed by exec()).