<html><head></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">
##

                x86 Memory: Addresses, Segmentation, Paging, etc

##

- In x86, there are three kinds of addresses:
    1. Logical - segment:offset
    2. Linear (aka Virtual) - 32 bits - 0 to 2^32 - 1
    3. Physical - the actual hardware - usually represented by 32 or 36 bits

+---+
| Memory Management Unit (MMU) |
| Logical ==> [Segmentation Unit] == Linear ==> [Paging Unit] ===> Physical
|
+---+

SEGMENTATION
~~~~~~~~~~~~

- Addresses are made up of a segment and offset (denoted - segment:offset)
- The segment specifies a location in linear memory, and the offset is an
  offset from that location.
- In real mode, the linear address is computed by: segment * 16 + offset
- In protected mode, the segment is a 16-bit field that points to an entry in
  either a Global Descriptor Table (GDT) or a Local Descriptor Table (LDT)

- The segment is stored in segment registers:
    - cs - code segment (current instruction is cs:EIP)
    - ds - data segment
    - ss - stack segment
    - es, gs, etc

- Segment format (protected mode):
    15   3 2   0
    [Index|TI|RPL]

    - TI (Table Indicator) specifies to look in the GDT or LDT
    - Index is an index into either the GDT or LDT
    - RPL (Requester Privilege Level) specifies the execution privileges when
      the cs register is set to that segment

- How does the CPU know where the GDT and LDT are?
    - The gdtr and ldtr registers hold the location and length of the GDT and
      LDT, respectively

- There is usually only one GDT, but processes may have their own LDT. See the
   modify_ldt() system call.

- Entries in the GDT and LDT are called Segment Descriptors

- To save time, whenever a segment register is changed, the CPU loads the
  segment descriptor into an internal register so it doesn't have to access
  the GDT or LDT every time it needs to convert a logical address to a virtual
  address

- Linux sets most segments to the full 0 to 2^32 - 1 linear memory range,
  so that the offset coincides with the linear address.

Segment Descriptors
~~~~~~~~~~~~~~~~~~~

Segment descriptors have the following fields:

BASE - The linear address of the first byte of the segment

G - Granularity Bit - If 0, the size is in bytes. If 1, the size is in multiples of 4096K

LIMIT - The offset of the last cell in the segment (cell size determined by G)

S - System Flag - Clear iff the segment stores critical data structres (such as an LDT or GDT). So 0 => critical, 1 => non-critical.

Type - The type of descriptor (see the following list)

P - Present

D or B

AVL - Ignored by Linux

Segment Descriptor Types:

Code Segment Descriptor:
- S flag set (meaning non-critical)
- Can be in LDT or GDT

Data Segment Descriptor:
- S flag set
- Can be in LDT or GDT
- Stack segments are these

Task State Segment Descriptor:
- Segment used to store processor registers
- S flag clear (meaning critical)
- Can only appear in the GDT

(see page 39 of Understanding the Linux Kernel for the layouts)

PAGING
~~~~~~~~~~~~~~~~~~~~~~~~

After the logical address has been converted to a linear address, the next step is paging, which converts the linear address to the physical address.

- Paging is enabled by setting the PG flag of the control register cr0.
- When PG = 0, the linear address is used directly as the physical address.

The 32 bits of linear memory are divided into 4KB pages. A 32 bit address is broken down into:
- Directory - The most significant 10 bits
- Table - The next 10 bits
- Offset - The remaining 12 bits (4KB)

- Directory bits are used as an index into the Page Directory, which points to a Page Table.
- The Table bits are used as an index into that Page Table, which contains a physical memory address (the start of a Page Frame).
- The offset is is added to that address (an offset into the Page Frame).

- ** The physical location in memory that can store a PAGE is called a PAGE FRAME. **

- Physical address = PD[linear >> 22][linear >> 12 &amp; 0x3FF] + linear &amp; 0xFFF
(a simplification)
~~~~~~~~~~~~~~~~~~~~~~~~

- Every process has its own Page Directory and Page Tables
    - Allows the kernel to keep processes separate even though they think they have the full 32-bit linear address space available to them.

- The cr3 register stores the address of the current Page Directory

- Page Directories and Page Tables have the same fields:
    - Present flag - 1 iff the page is in physical memory (0 could mean it has been swapped out to disk or something)
    - Most significant 20 bits of the address of the start of the page frame
        - Page frames are on 4096-byte (2^12) boundries, so the least significant 12 bits are always 0
    - Accessed flag
    - Dirty flag
    - Read/write flag
    - User/Supervisor flag
    - PCD and PWT flags
    - Page Size flag - set if the page frame is large (see extended paging)
    - Global flag

- If a process tries to access a page with the present flag clear, the CPU generates a Page Fault exception.

- Terminology: "page tables" refer to all kinds of lookup tables used during paging: Page Directories, Page Tables, etc. "Page Tables" refer to the actual "Page Table" object that a Page Directory entry points to.

Extended Paging

- Page frames are 4MB instead of 4KB when the Page Size flag is set.
- Page Tables are not needed for large pages. The linear address is broken into
    - Directory - most significant 10 bits - index into the Page Directory
    - Offset - the remaining 22 bits - offset into the 4MB page

- Extended paging can be used at the same time as regular paging.

Physical Address Extension (PAE)

- Old systems used 32-bit physical addresses, which limits them to 4GB of RAM
- Intel CPUs after Pentium Pro support PAE, which widens the physical address to 36 bits (64GB).
- Activated by the PAE flag in the cr4 register

- The paging mechanism is different:
    - Large pages (see above) are 2MB not 4MB when PAE is enabled
    - Page table entries are twice as large (up to 64 bits from 32 bits)
        - because the physical address field in the entries increased from 20 bits to 24 bits
    - adds another level of page tables: Page Directory Pointer Table (PDPT) comes before the Page Directory

    Normal paging with PAE:
        - cr3 points to a PDPT
        - bits 30-31 index the PDPT (PDPTs have 4 entries)
        - bits 29-21 index the Page Directory
        - bits 20-12 index the Page Table
        - bits 11-0 are an offset into the page
    Extended paging with PAE:
        - cr3 points to a PDPT
        - bits 30-31 index the PDPT

        - bits 29-21 index the Page Directory
        - bits 20-0 are an offset into the 2MB page

- Even with PAE, linear addresses are still only 32 bits, so a process can
  access at most 4GB of RAM. You have to change the cr3 register to point to a
  different Page Directory Pointer Table to access more RAM.

64-bit Paging
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    - With 64-bit linear addresses, we need more paging levels
    - On x86_64, an address is broken into 5 pieces (4 paging levels + offset):
        9 bits, 9 bits, 9 bits, 9 bits, 12 bits

Page Table Caching
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- The Translation Lookaside Buffer (TLB) in the CPU is used to cache
  conversions from linear addresses to physical addresses (because checking
  multiple levels of page tables every time would be very slow).
- Each CPU has its own TLB.

THE LINUX PAGING MODEL
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- Linux uses an abstraction of the paging model that works on multiple
  architectures that have different levels of paging. It generalizes the paging
  scheme to four levels of page tables:
    - Page Global Directory (PGD)   <-- broadest
    - Page Upper Directory (PUD)
    - Page Middle Directory (PMD)
    - Page Table (PT)   <-- most specific

- On architectures that don't need all 4 paging levels (like x86) Linux
  "eliminates" some of them, by:
    - saying that 0 bits of the linear address are used to index that table
    - "folding them back" onto their successor

- 2 paging levels (x86): PGD --> PT (PUD and PMD are "folded back" to the PGD)
- 3 paging levels (x86 with PAE): PGD --> PUD ---> PT (PMD is "folded back" to
  the PUD)

For example, in non-PAE x86:
    - The PGD is 10 bits of the linear address
    - The PUD is 0 bits
    - The PMD is 0 bits
    - The PT is 10 bits
    - The offset is 12 bits

    - The function pud_offset(pgd, addr), which normally takes the address of a
      PGD entry and a linear address, and returns the address of the PUD entry
      corresponding to the linear address, instead just returns the address of
      the PGD entry passed in.
        - In pseudocode...
            pud_offset(pgd, addr) {
                return pgd;
            }
            ... which can be optimized out by the compiler

    - The same thing happens for other functions that look up an entry in a
      "folded back" page table (PUD and PMD).

    - These lookup functions are only used by the kernel to generate and
      maintain the lookup tables. The actual lookup is done in hardware (when a
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

user mode process reads/writes to/from memory).

- Don't be confused: The "folded back" page tables are **NOT** single-entry tables with that entry pointing to the next table. They really don't exist in memory at all. When you think you have a pointer to a PUD entry, it really is a pointer to a PGD entry.

Process & Kernel Page Tables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- Linear address space of a process is divided into two regions:
  - 0x00000000 to 0xBFFFFFFF can be accessed in user mode
  - 0xC0000000 to 0xFFFFFFFF can only be accessed in kernel mode
  - The kernel lives in 0xC0000000 to 0xFFFFFFFF

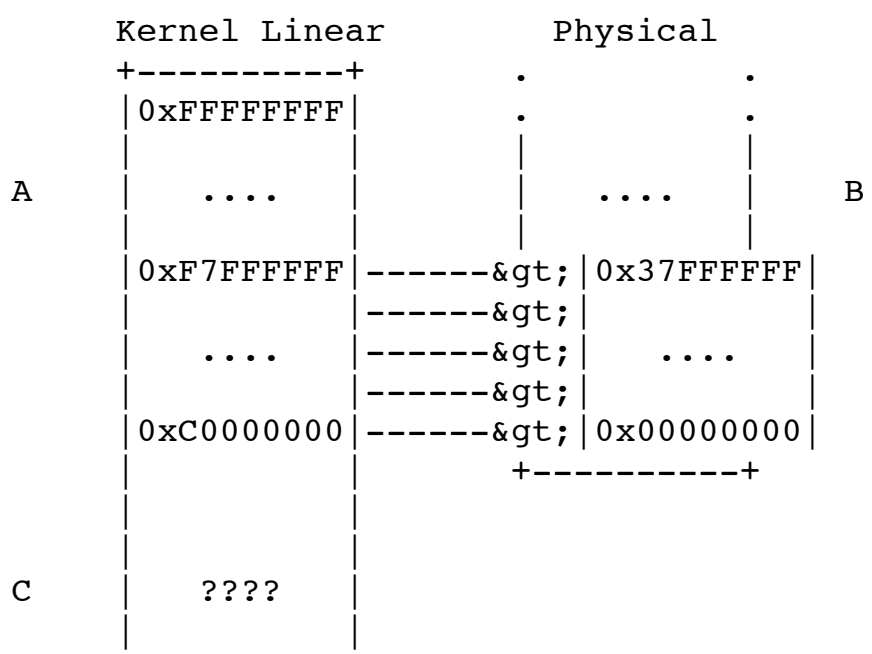
- So the first 768 of 1024 entries of the Page Global Directory (corresponding to the range 0x00000000 to 0xBFFFFFFF) depend on the current process, but the remaining entries (256 or 1024) (corresponding to 0xC0000000 to 0xFFFFFFFF) are always the same.

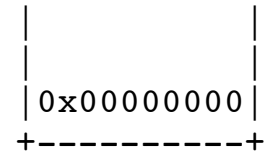
- The kernel has its own Page Global Directory, called the "master kernel Page Global Directory"
- The high 256 entries (0xC0000000 to 0xFFFFFFFF) of each process's PGD is the same as the master kernel PGD's

- The kernel sets up its page tables in two phases:
  - First, just enough to get out of real mode and initialize some data structures
  - Second, it re-does the page table initialization properly

Phase 1:
  - Supposing the kernel only needs 8MB of RAM initially, it maps linear address ranges 0x00000000 to 0x007FFFFF and 0xC0000000 to 0xC07FFFFF to the physical address range 0x00000000 to 0x007FFFFF.
  - This lets it access the first 8MB using the same linear addresses in real mode and protected mode.

Phase 2:
  - Phase 2 is basically the same as Phase 1, except that it maps everything above 0xC0000000 onto physical addresses starting at 0x00000000.
  - The User/Supervisor flag of the resulting PGD entries are clear so that user mode processes can't access the kernel address space.
  - If the total physical RAM size is less than 896MB, the kernel can now access all of it using linear addresses from 0xC0000000 to 0xF7FFFFFF
    - The high 128MB are reserved for special mappings.
    - (0xFFFFFFFF - 0xC0000000)bytes - 128MB = 896MB
  - If RAM is more than 896MB Linux will have to change some of the page table entries to be able to access it.


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



A: The top 128MB is reserved for special kinds of mappings (see the next section).

B: The kernel must change its page table entries to be able to access physical memory above 0x3FFFFFFF.

C: The mappings from 0x00000000 to 0xC0000000 depend on the current process.

Fix-Mapped Linear Addresses

- The top 128MB of the kernel's linear address space is used for noncontiguous memory allocation and fixed-mapped linear addresses.

- A fix-mapped linear address is just a linear address mapped to an arbitrary physical address.

- Used as an alternative to pointer variables:
    - e.g. just hard-code the address 0xFF001234 and use fix-mapping to map it onto the physical address it should point to.

</pre></body></html>