Memory Mapping in 64 Bit Mode and Registers

Chapter 4

Page

- What is page?

Page

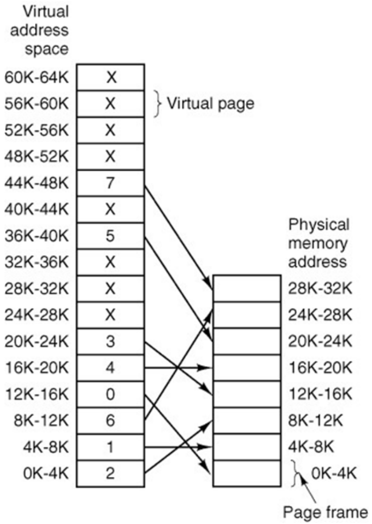
- What is page?
  - A page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table.
  - It is the smallest (usually) unit of data for memory management in a virtual memory operating system.
  - In our case we will consider 4kb pages.

Page

- What is page?
  - A page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table.
  - It is the smallest (usually) unit of data for memory management in a virtual memory operating system.
  - In our case we will consider 4kb pages.
  - Check with `getconf PAGESIZE`.
- How is the mapping of a page between virtual address space and physical address space done?

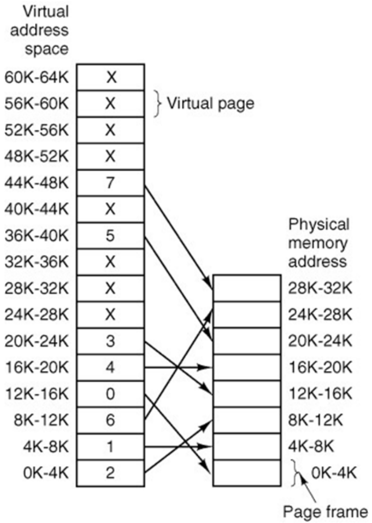
Single level paging

- To the right we have an example of a single layer paging system
- The X's indicate that the virtual page does not at present have a corresponding physical page.
    - If a program requires a virtual page marked with an X this causes a page fault.
    - The system must then allocate some physical memory to assign this virtual page to.
        - But what if there is no space left?



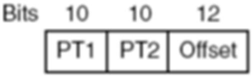
Single level paging

- The primary problems with a single level paging system is the size of the table.
  - Consider the linux process model with a virtual memory space of 128TB that would imply a table with $2^{(47-12)} = 2^{35}$ entries.
  - About 34 billion entries!
  - Assuming a 1GB of physical memory we would need to address $2^{(30-12)} = 2^{18}$ physical 4k pages.
    - So we would need 18-bits plus 1 bit to indicate if the entry is valid or invalid.
  - Just this table will occupy $2^{35} * 19$ bits, which is about 81.6 GB.



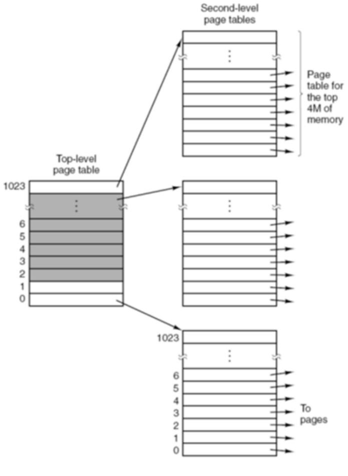
Multilevel Page Tables

- To get around the problem of having to store huge page tables in memory all the time, many computers use a multilevel page table.
- As a simple example consider we have a 32-bit virtual address (4GB) ,that is partitioned into a
  - 10-bit PT1 field
  - 10-bit PT2 field,
  - and a 12-bit Offset field (for the 4k pages).
- In general if we used PT1+PT2 together we would be working 2^{20} pages
  - That's a lot of pages to keep in memory



Multilevel Page Tables

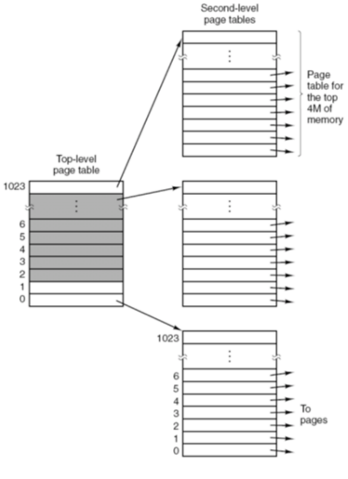
- The **top** level page table corresponds to the first 10-bits.
  - which maps to 1024 page tables
- The **second** level page table corresponds to the second 10-bits
  - Which maps to the 1024 pages of size 4K
- The last remaining 12 bits are used as the offset to address the contents of the 4K page.



(b)

Multilevel Page Tables

- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
  - In particular, those that are not needed should not be kept around.
- By marking elements of the top-level page table as absent we do not need to maintain (or store) all the second level page tables, saving substantially on space.



(b)

Memory mapping pages and tables

- Linux uses 4 layer page tabling.
- Each page is $2^{12} = 4096$ bytes
- An address is 8 bytes (not all used)
- Each page table can hold $2^{9} = 512$ addresses
- A 9 bit field is needed to index the mapping tables
- Current mapping uses 48 bits, so we are limited to 2^{48} bytes which is 256 TB

Logical memory address fields

63–48	47–39	38–30	29–21	20–12	11–0
unused	PML4 index	page directory pointer index	page directory index	page table index	page offset

- Bits 47-39 are used to index the PML4 table
- Bits 38-30 are used to index the selected page directory pointer table
- Bits 29-21 are used to index the selected page directory table
- Bits 20-12 are used to index the selected page table
- Bits 11-0 are the offset into the page (for 4 KB pages)

Large pages

- Using the first 3 existing levels of page tables, we can have large pages with $2^{21} = 2097152$ bytes.
- This is used by Linux for the kernel

CPU support for fast lookups

- A CPU uses a special cache called a "Translation Lookaside Buffer" or TLB to speed up memory translation
- A TLB operates much like a hash table
- Presented with a logical address, it produces the physical address or failure in about 1/2 a clock cycle
- The Intel Core i7 has 2 levels of TLBs
  - Level 1 holds 64 small page translations (or 32 big pages)
  - Level 2 holds 512 page translations
  - Large programs with small pages will experience TLB misses which can be satisfied fairly rapidly with normal cache
  - Very large programs can crawl

Chapter 5

Register basics

- Computer main memory has a latency of about 80 nanoseconds
- A 3.3 GHz CPU uses approximately 0.3 nsecs per cycle
- Memory latency is about 240 cycles
- The Core i7 has 3 levels of cache with different latencies
  - Level 3 about 48 nsec latency or about 150 cycles
  - Level 2 about 10 nsec latency or about 39 cycles
  - Level 1 about 4 nsec latency or about 12 cycles
- There is a need for even faster memory
- This ultra-fast "memory" is the CPU's registers
- Some register-register instructions complete in 1 cycle

x86-64 registers

- CPUs running in x86-64 mode have 16 general purpose registers
- There are also 16 floating point registers (XMM0-XMM15)
- There is also a floating point register stack which we ignore
- The general purpose registers hold 64 bits
- The floating point registers can be either 128 or 256 bits
  - The CPU can use them to do 1 32 bit or 1 64 bit floating point operation in an instruction
  - The CPU can also use these to do packed operations on multiple integer or floating point values in an instruction
  - "Single Instruction Multiple Data" - SIMD
- The CPU has a 64 bit instruction pointer register - `rip`
  - contains the address of the next instruction to execute.
- There is a 64 bit flags register, `rflags`, holding status values like whether the last comparison was positive, zero or negative

General purpose registers

- These registers evolved from 16 bit CPUs to 32 bit mode and finally 64 bit mode
- Each advance has maintained compatibility with the old instructions
- The old register names still work
- The old collection was 8 registers which were not entirely general purpose
- The 64 bit collection added 8 completely general purpose 64 bit registers named `r8` - `r15`

The 64 bit registers evolved from the original 8

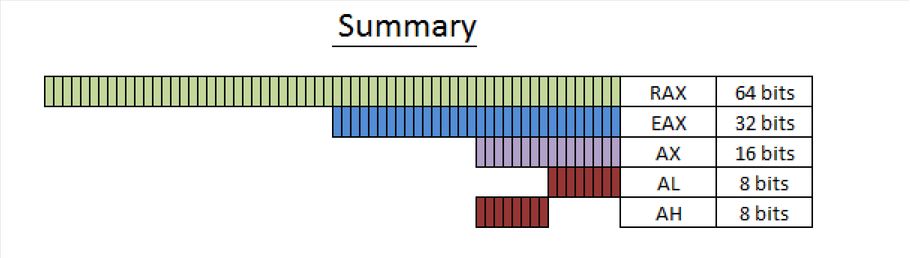
- Software uses the “r” names for 64 bit use, the “e” names for 32 bit use and the original names for 16 bit use
- `rax` - general purpose, accumulator
  - `rax` uses all 64 bits
  - `eax` uses the low 32 bits
  - `ax` uses the low 16 bits
- `rbx`, `ebx`, `bx` - general purpose
- `rcx`, `ecx`, `cx` - general purpose, count register
- `rdx`, `edx`, `dx` - general purpose
- `rdi`, `edi`, `di` - general purpose, destination index
- `rsi`, `esi`, `si` - general purpose, source index
- `rbp`, `ebp`, `bp` - general purpose, stack frame base pointer
- `rsp`, `esp`, `sp` - stack pointer, `rsp` is used to push and pop

The original 8 registers as bytes

- Kept from the 16 bit mode
  - `al` is the low byte of `ax`, `ah` is the high byte
  - `bx` can be used as `bl` and `bh`
  - `cx` can be used as `cl` and `ch`
  - `dx` can be used as `dl` and `dh`
- New to x86-64
  - `dil` for low byte of `rdi`
  - `sil` for low byte of `rsi`
  - `bpl` for low byte of `rbp`
  - `spl` for low byte of `rsp`
- There is no special direct way to access any "higher" bytes of registers

Visual summary

Break down of a 64-bit register.



The 8 new general purpose registers as smaller registers

- Here the naming convention changes
- Appending “d” to a register accesses its low double word - `r8d`
  - double word $=$ 4 bytes $=$ 32 bits.
- Appending “w” to a register accesses its low word - `r12w`
  - single word $=$ 2 bytes $=$ 16 bits.
- Appending “b” to a register accesses its low byte - `r15b`

Moving a constant into a register

- Moving is fundamental
- `yasm` uses the mnemonic `mov` for all sorts of moves
- The generated code from `gcc` uses mnemonics like `movq`
  - gcc uses AT&T syntax by default
- Most instructions can use 1, 2 or 4 byte immediate fields
- `mov` can use an 8 byte immediate value.

```
mov rax, 0x0123456789abcdef ; can move 8 byte immediates
mov rax, 0
mov eax, 5                  ; the upper half is set to 0
mov r8w, 16                 ; affects only low word
```

Moving a value from memory into a register

```
    segment .data
a       dq      175
b       dq      4097
c       db      1, 2, 3, 4, 5, 6, 7, 8
d       dd      0xffffffff
    segment .text
    mov     rax, a
    mov     rbx, [a]
    mov     rcx, [c]
    mov     edx, [d]
```

- Using simply **a** places the address of **a** into **rax**
- Using **[a]** places the value of **a** into **rbx**

A program to add 2 numbers from memory

```
    segment .data
a       dq      175
b       dq      4097
    segment .text
    global  main
main:
    mov     rax, [a]    ; mov a into rax
    add     rax, [b]    ; add b to rax
    xor     rax, rax
    ret
```

Move with sign extend or zero extend

- If you move a double word into a double word register (e.g. eax), the upper half is zeroed out
- If you move a 32 bit immediate into a 64 bit register it is sign extended
- Sometimes you might wish to load a smaller value from memory and fill the rest of the register with zeroes
- Or you may wish to sign extend a small value from memory
- For `movsx` and `movzx` you need a size qualifier for the memory operand

```
movsx  rax, byte [data]      ; move byte, sign extend
movzx  rbx, word [sum]       ; move word, zero extend
movsxd  rcx, dword [count]    ; move dword, sign extend
```

Moving values from a register into memory

- Simply use the memory reference as the first operand

```
mov    [a], rax        ; move a quad word to a
mov    [b], ebx        ; move a double word to b
mov    [c], r8w        ; move a word to c
mov    [d], r15b       ; move a byte to d
```

Moving data from one register to another

- Use 2 register operands

```
mov     rax, rbx     ; move rbx to rax
mov     eax, ecx     ; move ecx to eax, zero filled
mov     cl, al       ; move al to cl, leave rest of
; unchanged
```