Write Your Own (minimal) Operating System Kernel

Jan Hauffa

(jhauffa@gmail.com)

Outline

1. Tools
2. Basic i386 System Programming
3. Implementation of a Minimal OS Kernel
4. Debugging

1. Tools

Tools

- gcc, binutils, qemu, xorriso
  - Windows: Cygwin, MinGW
  - Mac: Xcode, DIY, MacPorts
- gcc cross compiler toolchain
- GRUB2 or BURG bootloader

Why Build a Cross Compiler?

- gcc and binutils are usually „tuned“ for a specific target OS:
  - binary format: Linux = ELF, MacOS = Mach-O, Windows = PE-COFF
  - features that require runtime support: stack protection, stack probing, …
- Run „gcc -v“ to see what your system compiler drags in.
- We need a „clean“ toolchain that can generate a specific binary format.

Building the Cross Compiler

- Get the latest versions of *gmp*, *mpfr* from http://ftp.gnu.org/gnu/ and *mpc* from http://www.multiprecision.org/
  - Build and install into a local prefix, I use /usr/local/cc
- Get the latest gcc-core from ftp.gnu.org.

```
1. mkdir build-gcc && cd build-gcc
2. ../gcc-XXX/configure --target=i586-elf
       --prefix=/usr/local/cc --disable-nls
       --enable-languages=c --without-headers
       --with-gmp=/usr/local/cc
       --with-mpfr=/usr/local/cc --with-mpc=/usr/local/cc
3. make all-gcc
4. sudo make install-gcc
5. make all-target-libgcc
6. sudo make install-target-libgcc
```

detailed instructions: http://wiki.osdev.org/GCC_Cross-Compiler

More boring stuff...

- If you're on Windows or MacOS, you'll need a „cross debugger“ that can handle ELF binaries.
  - Get the latest GDB from ftp.gnu.org.

```
./configure --target=i586-elf --prefix=/usr/local/cc
```

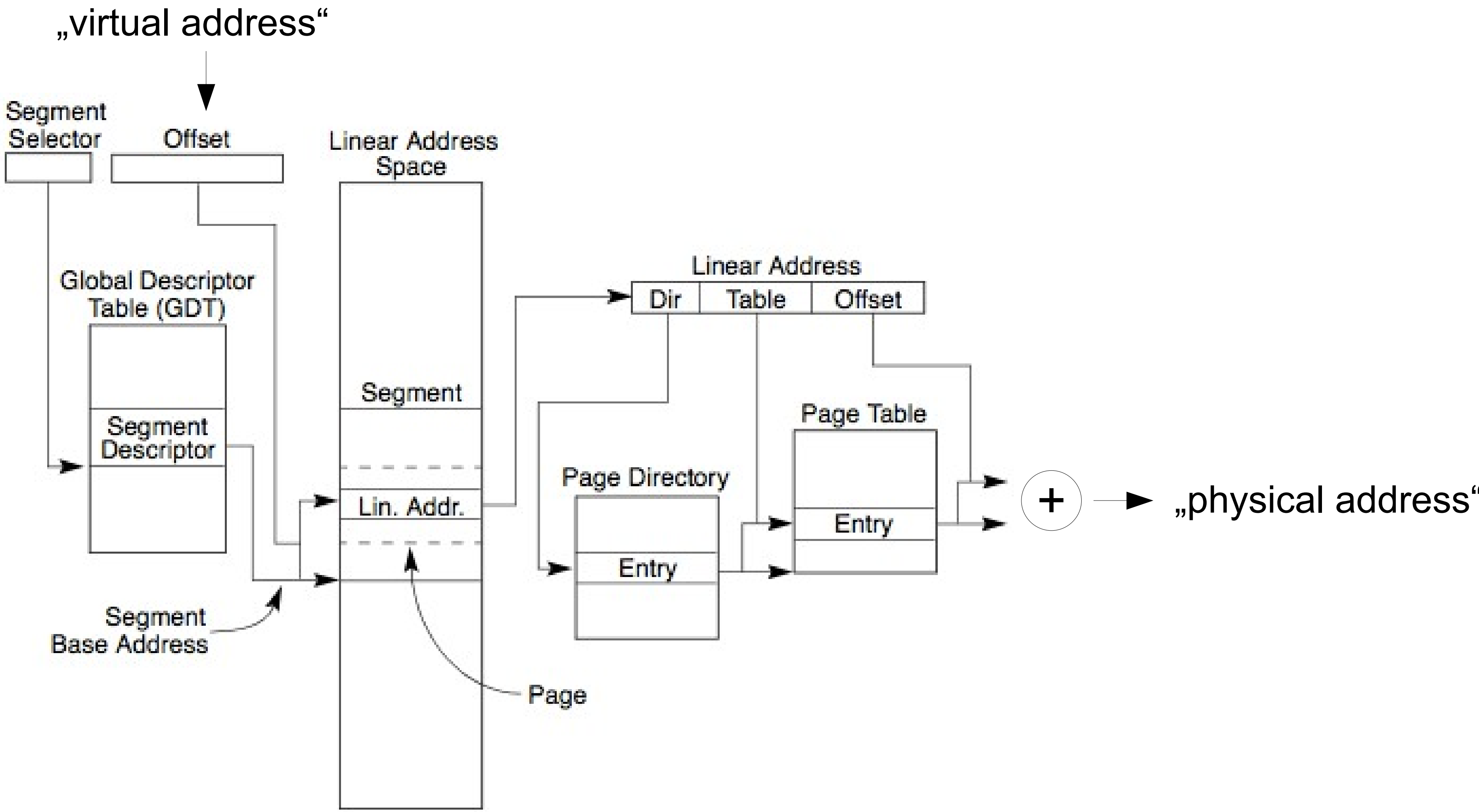
- We need a bootloader:
  - on Linux, build GRUB2
  - on Windows and MacOS, follow the instructions on https://help.ubuntu.com/community/Burg to build BURG
  - Run *grub-mkrescue* to build an ISO image.

2. Basic i386 System Programming

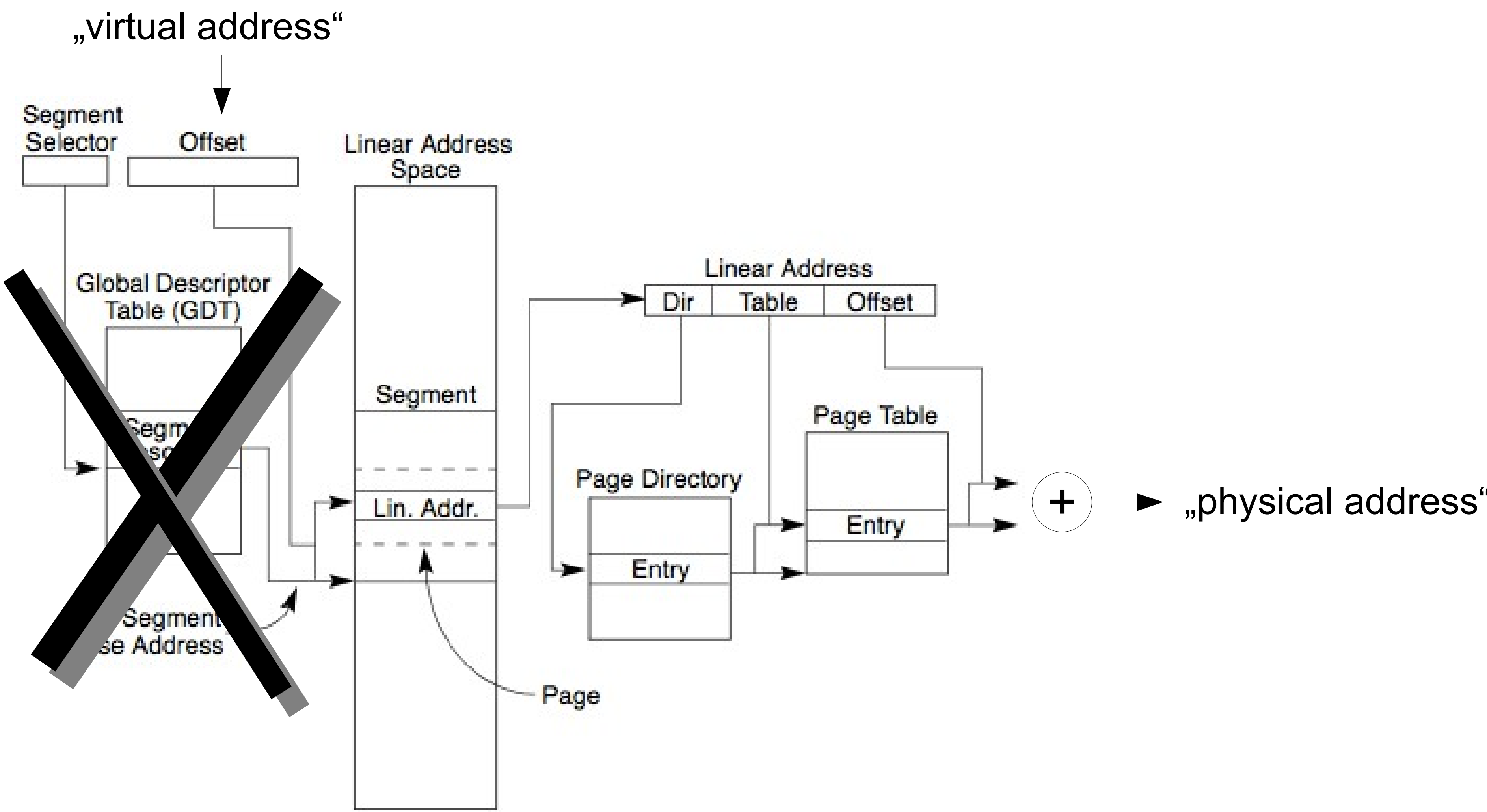
Operating Modes of an x86 CPU

- Real mode: introduced with the 8086 in 1978
  - 20-bit physical address space, no virtual memory
- 16-bit Protected Mode: 80286 in 1982
  - 24-bit physical and 30-bit virtual address space
- 32-bit Protected Mode: 80386 in 1985
  - 32-bit physical and virtual address space
- Long mode: AMD K8 in 2003 (AMD64), adopted by Intel (EM64T / Intel 64) in 2004
  - up to 52 bits of physical and 64 bits of virtual address space, currently both 48 bits

Address Translation in Protected Mode

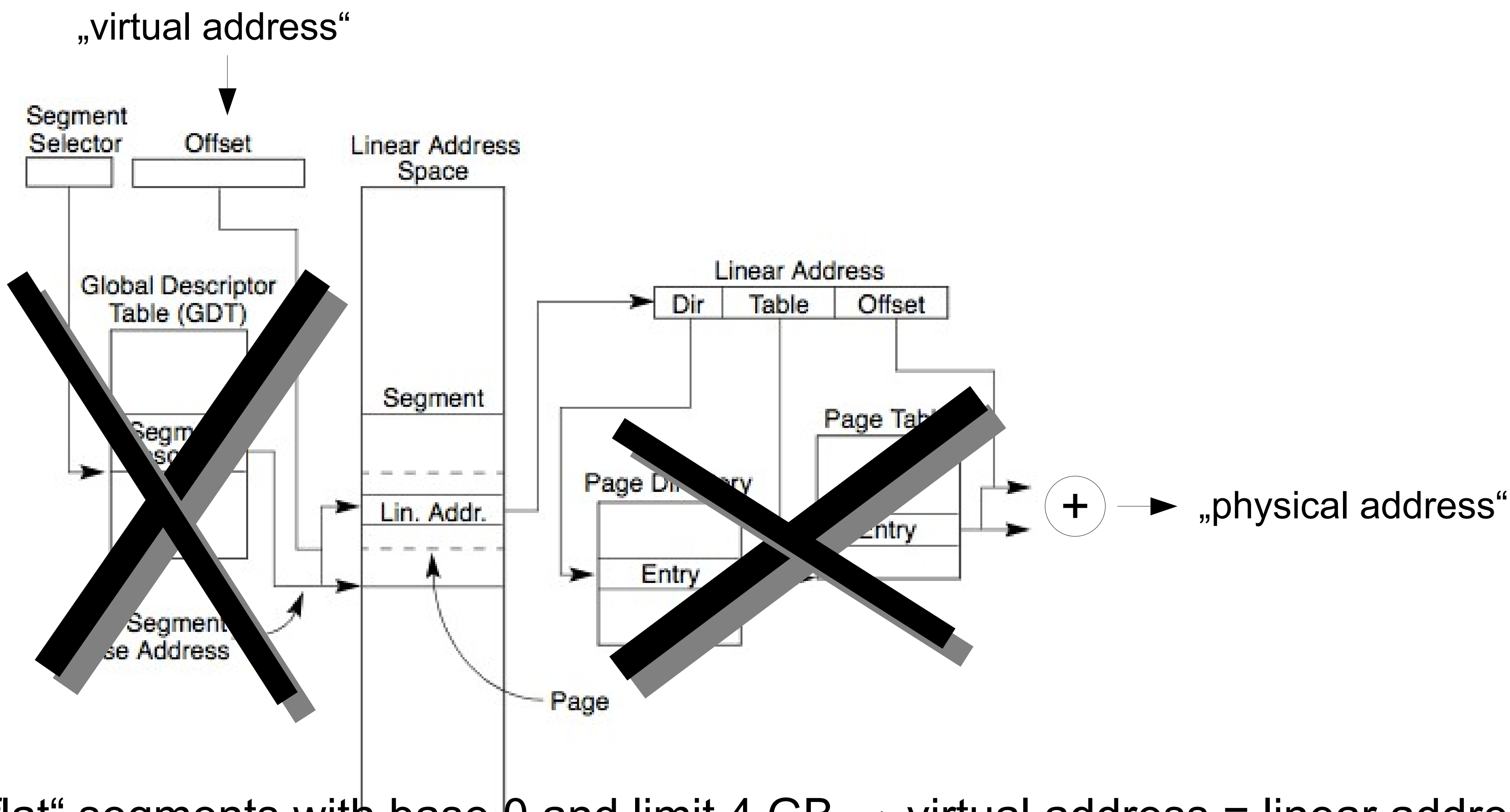


Address Translation in Protected Mode



„flat“ segments with base 0 and limit 4 GB → virtual address = linear address

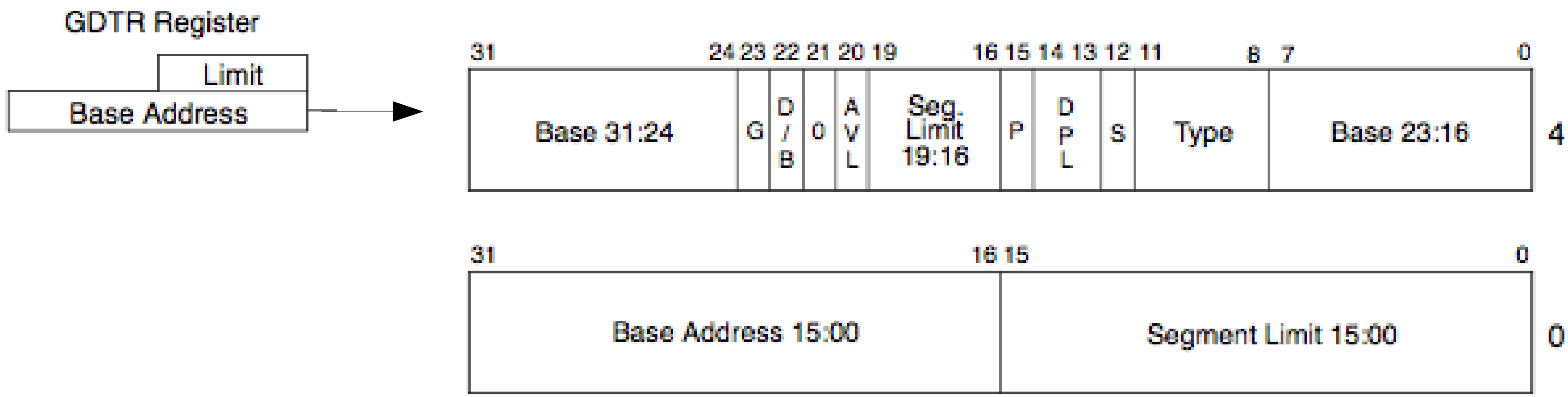
Address Translation in Protected Mode



„flat“ segments with base 0 and limit 4 GB → virtual address = linear address

paging disabled → linear address = physical address

Global Descriptor Table



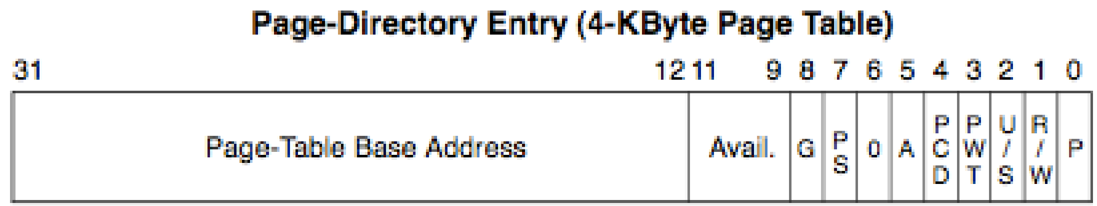
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

First descriptor in GDT is not used, selector 0 is always invalid!

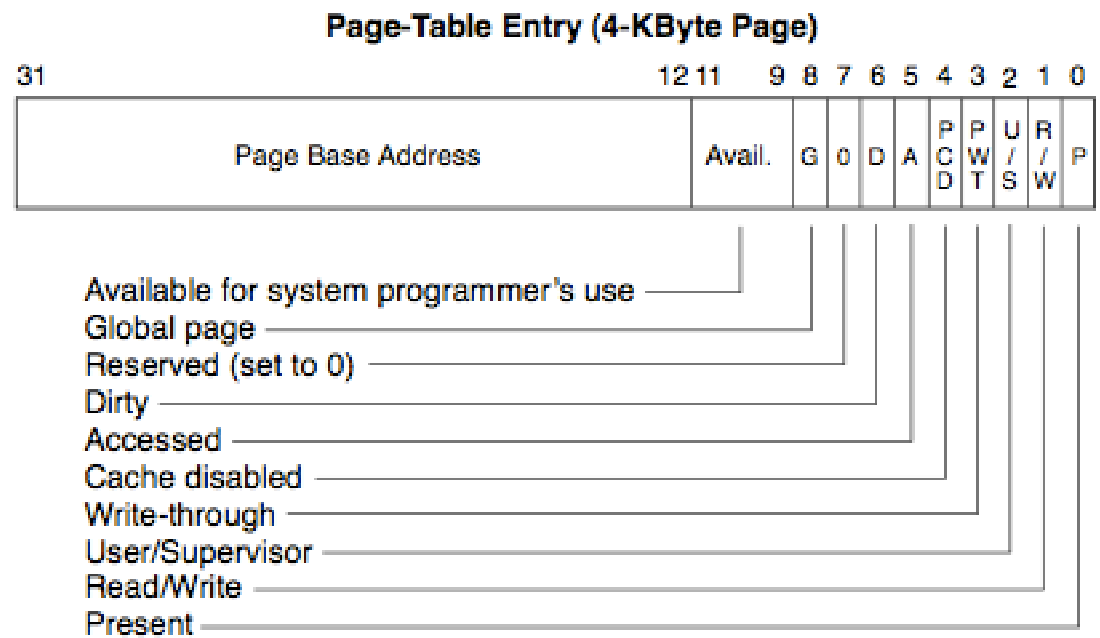
Page Tables

Physical address of Page Directory in register CR3

Page Directory = 1024x



Page Table = 1024x



Boot Sequence on a PC

1. CPU starts in real mode, executes instruction at 0xFFFF0 → top of BIOS ROM
2. self test (POST), configuration of devices (e.g. PCI resource allocation), initialization of BIOS extensions (option ROMs)
3. load first sector of boot device („boot sector“, MBR) to 0x7C00
   a. MBR relocates itself, loads first sector of active partition to 0x7C00
4. boot sector loads „second stage“ boot loader
   a. boot loader knows how to read kernel from file system
   b. usually loads kernel to an address > 1MB and switches to protected mode before jumping to kernel entry point

Multiboot Specification

- Goal: unified communication between boot loader and kernel
- reference implementation is GRUB
- covers:
  - kernel binary format
  - CPU state at kernel execution
  - passing of information from loader to kernel
- current version is 0.6.96, „Multiboot 2“ under development, but not completely implemented

Multiboot: Binary Format

- Multiboot header within the first 8KB of binary:

0	u32	magic	0x1BADB002
4	u32	flags	
8	u32	checksum	two's complement of magic+flags
12	u32	header_addr	
16	u32	load_addr	
20	u32	load_end_addr	physical addresses
24	u32	bss_end_addr	
28	u32	entry_addr	

- If binary format is ELF, load addresses can be omitted, are read from the program header table.

Multiboot: CPU State

- CPU in protected mode, paging disabled
- Interrupts disabled
- CS is valid code segment, all other segments and GDTR undefined
- stack undefined
- EAX contains magic 0x2BADB002
- EBX contains physical address of multiboot info structure

Multiboot: Info Structure

0	u32	flags	
4	u32	mem_lower	if flags[0] set
8	u32	mem_upper	if flags[0] set
12	u32	boot_device	if flags[1] set
16	u32	cmdline	if flags[2] set
20	u32	mods_count	if flags[3] set
24	u32	mods_addr	if flags[3] set
28 - 40		syms	if flags[4] or flags[5] set
44	u32	mmap_addr	if flags[6] set
48	u32	mmap_length	if flags[6] set
52 - 86		other optional fields	

Multiboot: mmap Structure

0	u32	size
4	u64	base_addr
12	u64	length
20	u32	type

add to entry base address
to get next entry

type: 1 = available RAM, all other values = do not touch

3. Implementation

Memory Layout

- Virtual address space divided into kernel space
  0xC0000000 – 0xFFFFFFFF
  and user space
  0x00000000 – 0xBFFFFFFF
  - Advantage: user applications are not dependent on size of kernel address space.

Linker Script

```
#define KERNEL_PHYS_BASE  0x00100000
#define KERNEL_VIRT_BASE  0xC0000000

OUTPUT_FORMAT("elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS
{
    . = KERNEL_VIRT_BASE + KERNEL_PHYS_BASE;
    kernel_start = .;

    .text . : AT(ADDR(.text) - KERNEL_VIRT_BASE)
    {
        *(.mbhdr)
        *(.text*)
    }

    [...]

    kernel_end = .;
}
```

Startup Code

- Problem: We link the kernel against base address 0xC0000000, but cannot load it at that address – not all systems have > 3GB RAM!
  - Load new GDT with segment base = 0x40000000 → by integer overflow we arrive at physical address!
- Set top of stack – 4KB reserved in BSS section.
- Push multiboot magic and pointer to info structure on stack.
- Call C main function.

Physical Memory Manager

- Problem: Cannot allocate space for data structures (page tables, etc.) without memory manager!
- Solution: Simple memory manager that uses static structures, build more complex manager on top.
- Bitmap of all physical memory, one bit for each 4KB page.
- Simple to implement, allows allocation of contiguous chunks (DMA!), but allocation is O(n).

Enabling Paging, Part 1

- Allocate physical memory for page directory and page tables.
- Set up page mappings as follows:
  - map kernel code / data from physical 0x100000 to virtual 0xC0000000
  - 1:1 mapping of kernel code / data
  - Highest page directory entry points to page directory itself → page tables appear in top 4 MB of virtual address space.
  - Map page directory right below page tables.

Enabling Paging, Part 2

- Modify GDT: set segment bases to 0.
- Load physical address of page directory into register CR3.
- Enable paging by setting bit 31 in register CR0.
- Reload segment registers and jump to 0xC0000000 + x
- Remove 1:1 mapping of kernel code / data.

Text Output

- VGA and compatible video cards have text mode „framebuffer“ at physical address 0xB8000.
- 80x25 characters, each character represented by 2 bytes in memory
  - character code (index into bitmap font)
  - attribute byte (FG/BG color)
- Map framebuffer into virtual address space, print „Hello, world!“

4. Debugging

Connecting GDB to qemu

- Build kernel with debugging info:

  `CFLAGS=-g make kernel`

- Run `qemu -S`, will wait for connection from debugger.
- In GDB (remember to use your cross debugger!):
  - `file objs/kernel.elf`
  - `target remote localhost:1234`

Debugging Tricks

- Debug-Port of qemu:
  `outb $`***`cc`***`, $0xe9`
  will print character ASCII *cc* on console
- Testing on a real PC:
  - Build GRUB with support for network booting and install on target PC.
  - Run TFTP server on development PC.
  - more details:
    http://wiki.osdev.org/GRUB#Load_your_kernel_over_network_.28TFTP.29

Literature References

- Intel IA-32 Manuals:
  http://www.intel.com/products/processor/manuals/index.htm
- Multiboot specification:
  http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
- Further information on OS development:
  http://www.acm.uiuc.edu/sigops/roll_your_own/
  http://wiki.osdev.org

Download the presentation slides and source code from
http://home.in.tum.de/~hauffa/