Loadable Kernel Modules

Overview

- What is a loadable kernel module
- When to use modules
- Intel 80386 memory management
- How module gets loaded in proper location
- Internals of module
- Linking and unlinking module

Kernel module description

- To add a new code to a Linux kernel, it is necessary to add some source files to kernel source tree and recompile the kernel. But you can also add code to the Linux kernel while it is running. A chunk of code added in such way is called a loadable kernel module
- Typical modules:
  - device drivers
  - file system drivers
  - system calls

When kernel code must be a module

- higher level component of Linux kernel can be compiled as modules
- some Linux kernel code must be linked statically then component is included in the kernel or it is not compiled at all
- **Basic Guideline**

  Build working base kernel, that include anything that is necessary to get the system up, everything else can be built as modules

Advantages of modules

- There is no necessity to rebuild the kernel, when a new kernel option is added
- Modules help find system problems (if system problem caused a module just don't load it)
- Modules save memory
- Modules are much faster to maintain and debug
- Modules once loaded are inasmuch fast as kernel

Module Implementation

- Modules are stored in the file system as ELF object files
- The kernel makes sure that the rest of the kernel can reach the module's global symbols
- Module must know the addresses of symbols (variables and functions) in the kernel and in other modules (/proc/syms <2.6 /proc/kallsyms - 2.6)
- The kernel keeps track of the use of modules, so that no modules is unloaded while another module or kernel is using it (/proc/modules)

Module Implementation

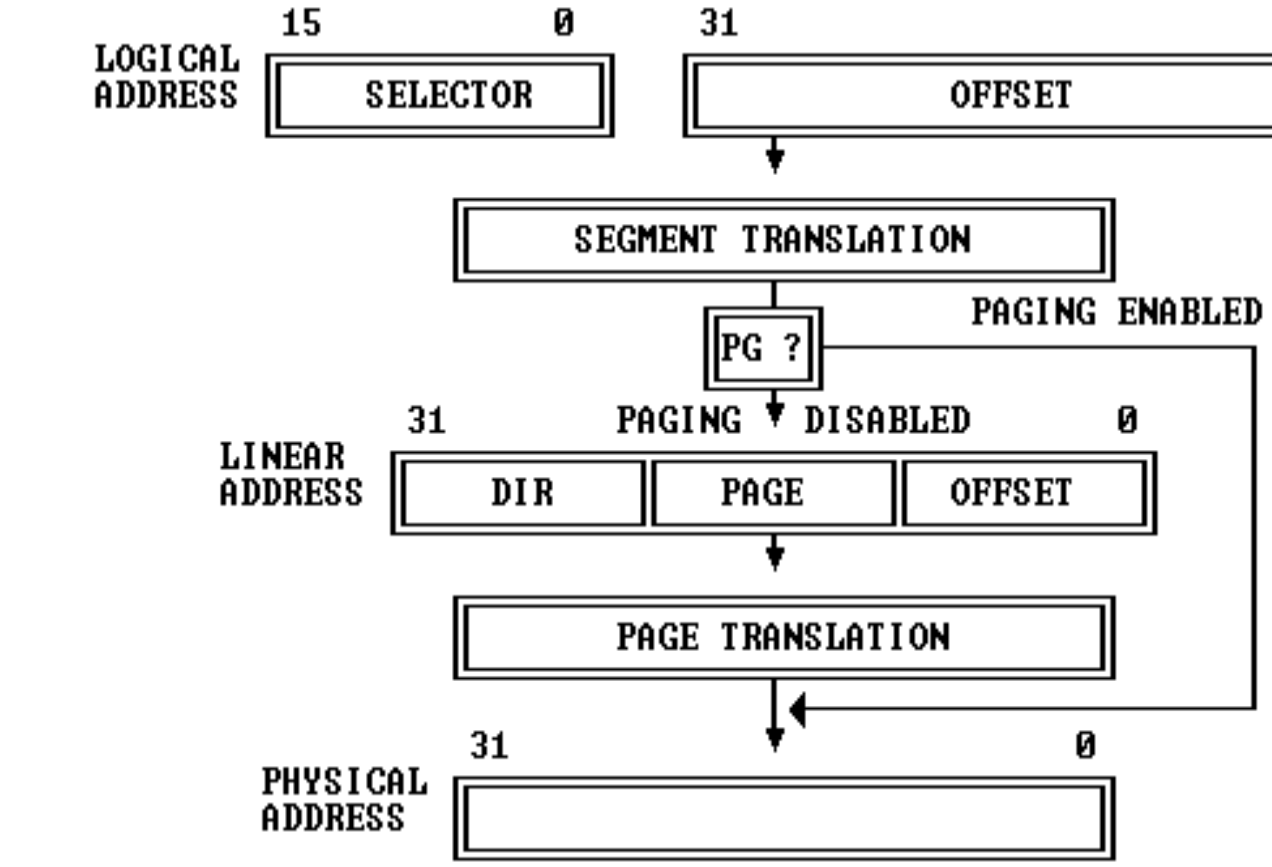
- The kernel considers only modules that have been loaded into RAM by the *insmod program and for each of them allocates memory area containing:*
  - *a module object*
  - *null terminated string that represents module's name*
  - *the code that implements the functions of the module*

Module Object

Type	Name	Description
unsigned long	size_of_struct	Size of module object
struct module *	next	Next list element
const char *	name	Pointer to module name
unsigned long	size	Module size
atomic_t	uc.usecount	Module usage counter
unsigned long	flags	Module flags
unsigned int	nsyms	Number of exported symbols
unsigned int	ndeps	Number of referenced modules
struct module_symbol *	syms	Table of exported symbols
struct module_ref *	deps	List of referenced modules
struct module_ref *	refs	List of referencing modules
int (*)(void)	init	Initialization method
void (*)(void)	cleanup	Cleanup method
struct exception_table_entry *	ex_table_start	Start of exception table
struct exception_table_entry *	ex_table_end	End of exception table

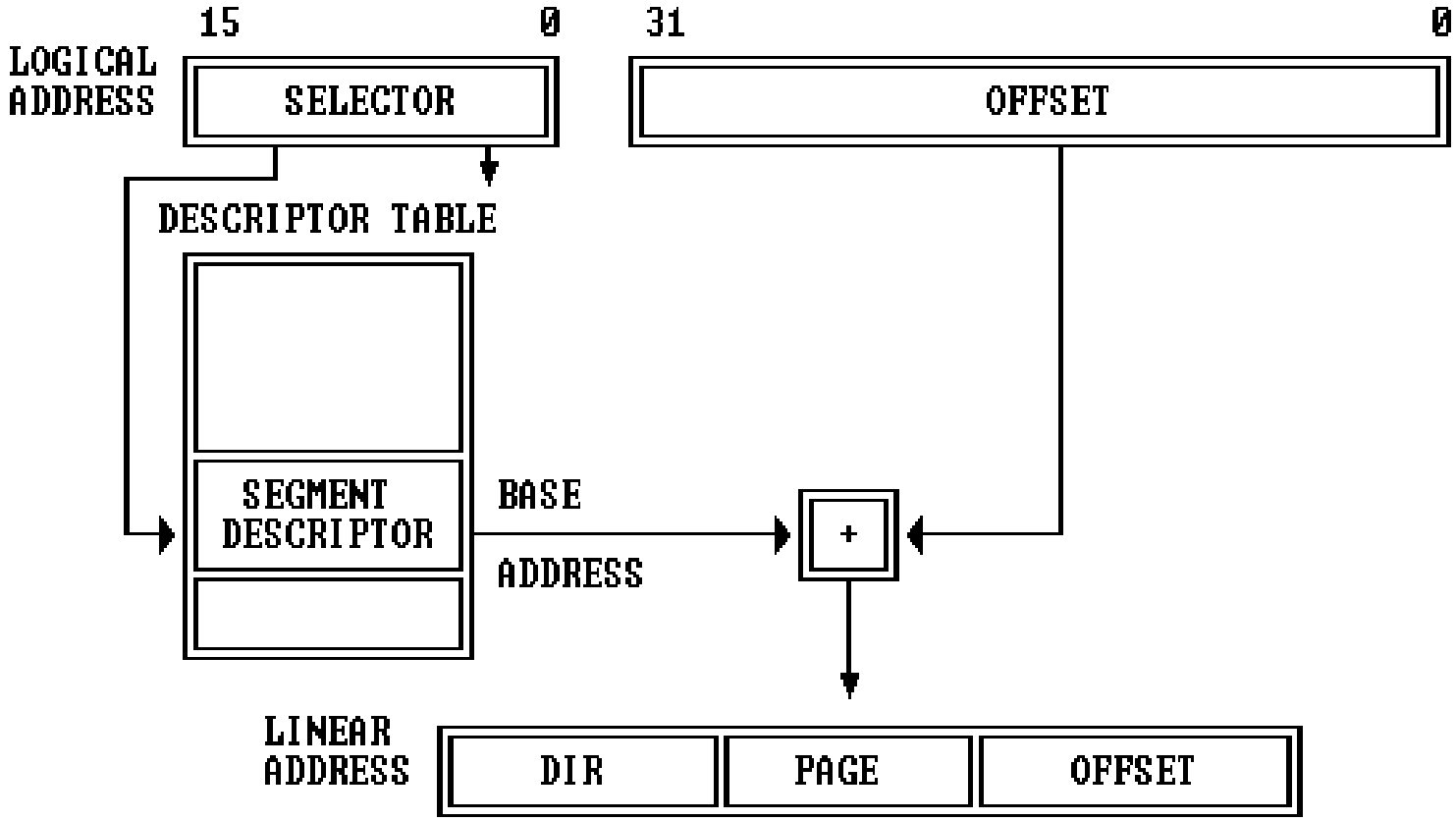
80386 Memory Management

Figure 5-1. Address Translation Overview



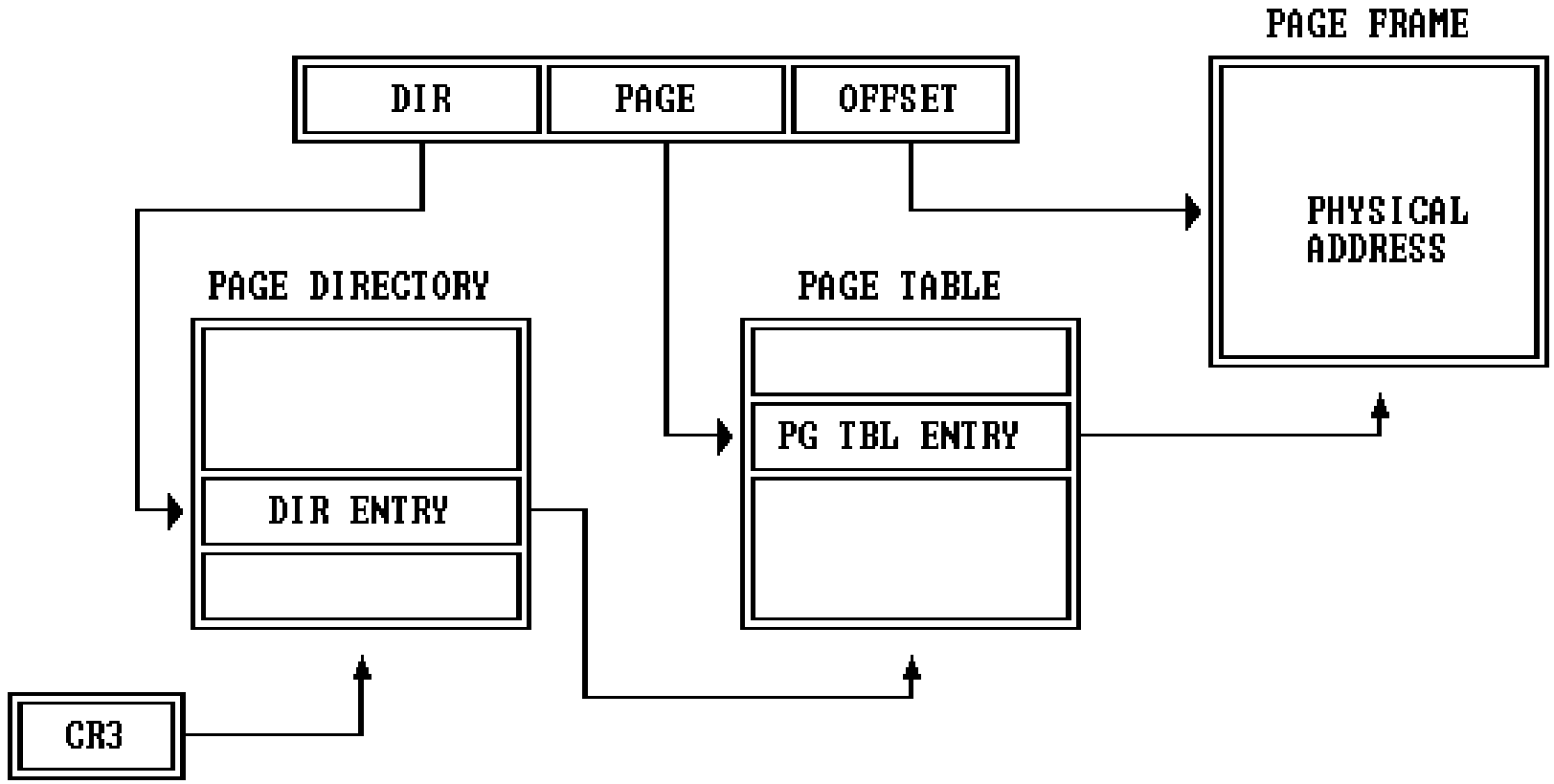
Segment Translation

Figure 5-2. Segment Translation

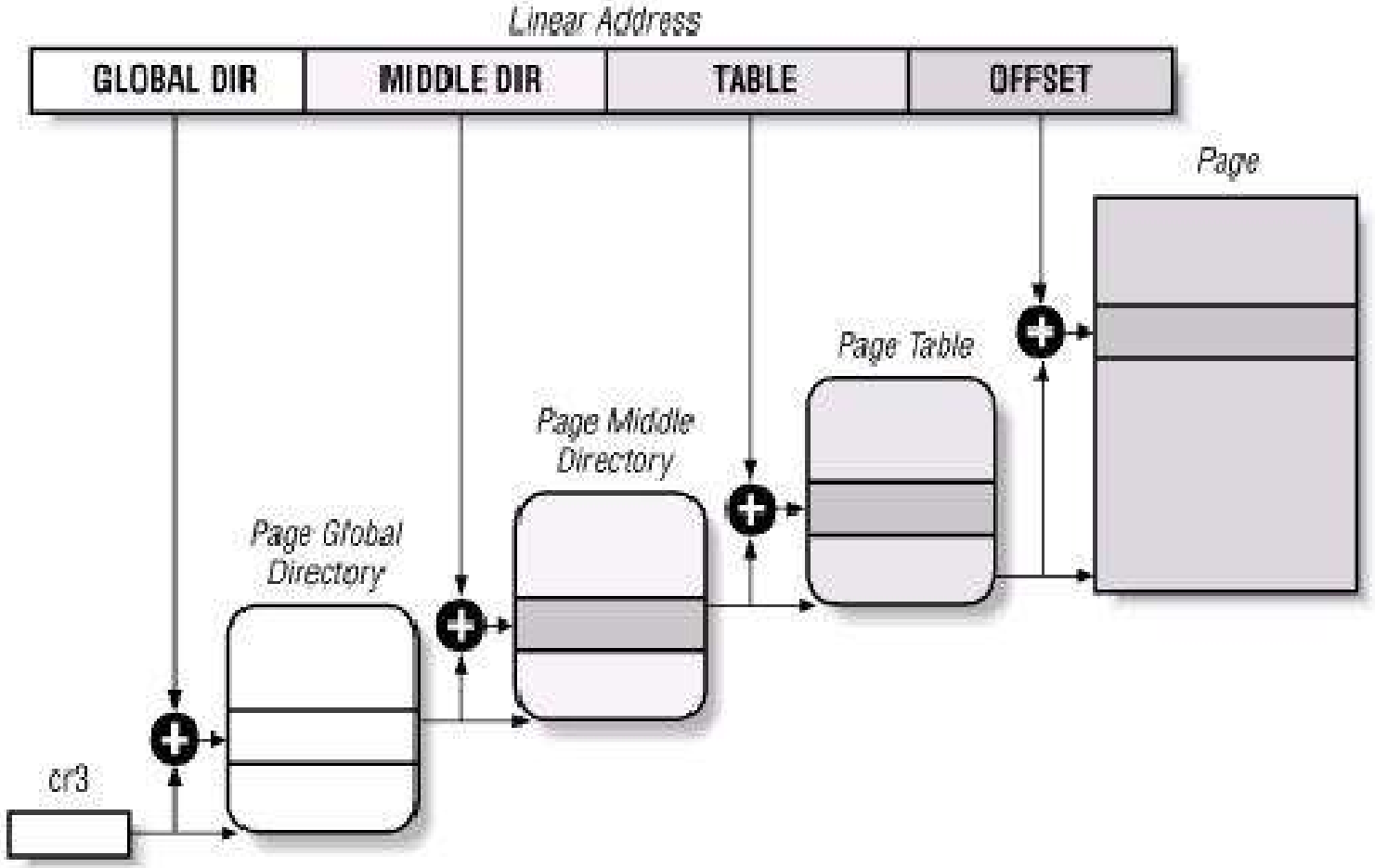


Page Translation

Figure 5-9. Page Translation

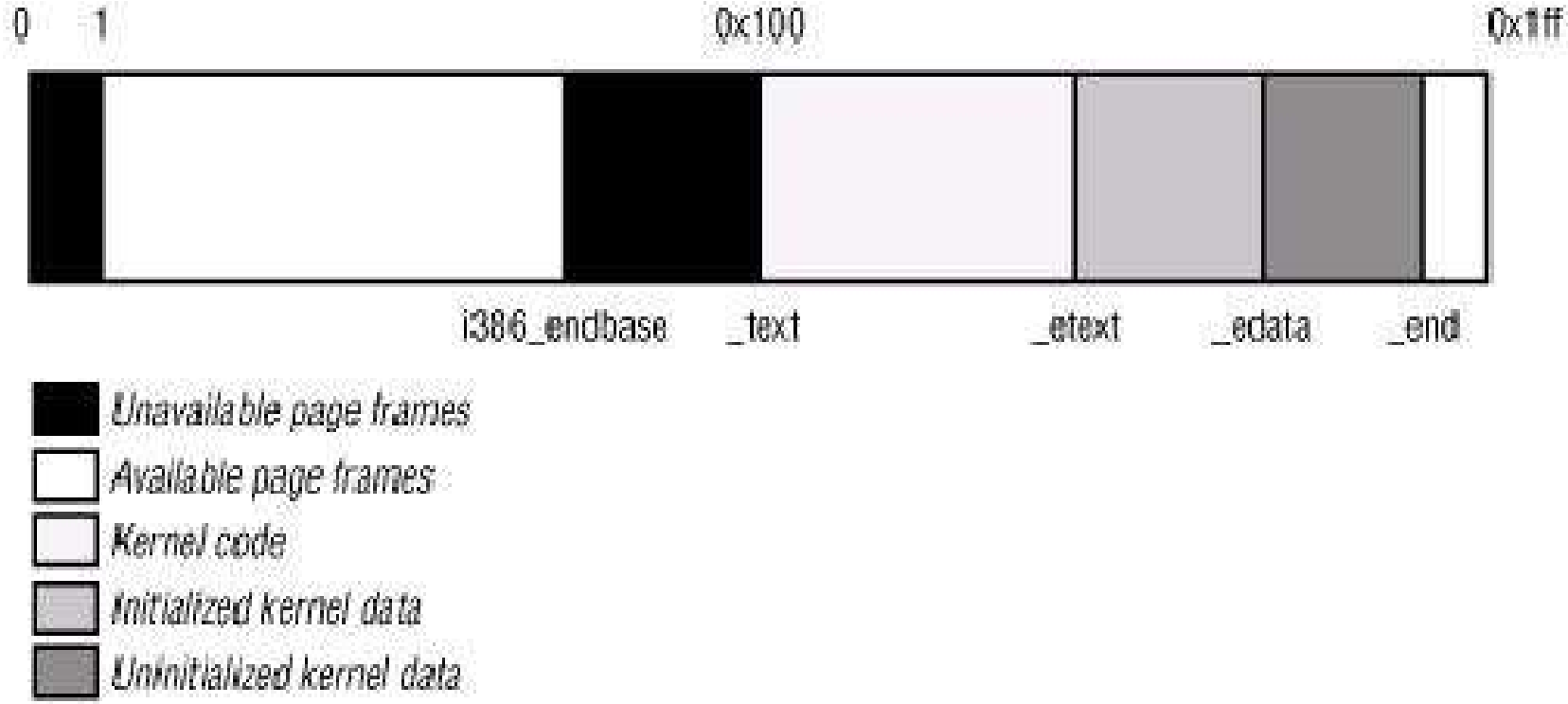


Linux paging model



Reserved Page Frames

. The first 512 page frames (2 MB) in Linux 2.2



Kernel Page Tables

- Provisional kernel page tables – first phase
  - The Page Global Directory and Page table are initialized statically during the kernel compilation.
  - During this phase of initialization kernel can address the first 4MB either with or without paging.
- Final kernel page table – second phase
  - Transforms linear addresses starting from PAGE_OFFSET into physical addressing starting from 0