Linux Kernel Analysis

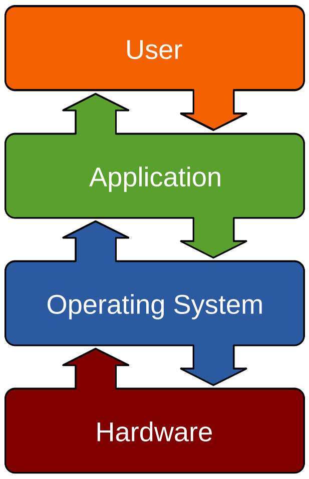
Wang Xiaolin
wx672ster+os@gmail.com

February 21, 2018

Contents

1 Overview

1.1 Basic Operating System Concepts



Two main objectives of an OS:

- Interact with the hardware components
- Provide an execution environment to the applications

Different OS, different ways

Unix hides all low-level details from applications

User mode vs. Kernel mode

MS-DOS allows user programs to directly play with the hardware components

Unix The original elegant design of the Unix system, along with the years of innovation and evolutionary improvement that followed, have made Unix a powerful, robust, and stable operating system. A handful of characteristics of Unix are responsible for its resilience. [*Linux Kernel Development*, Chapter 1, *Introduction to the Linux Kernel*]

- First, Unix is simple: Whereas some operating systems implement thousands of system calls and have unclear design goals, Unix systems typically implement only hundreds of system calls and have a very clear design.
- Next, in Unix, *everything is a file*. This simplifies the manipulation of data and devices into a set of simple system calls: `open()`, `read()`, `write()`, `ioctl()`, and `close()`.
- In addition, the Unix kernel and related system utilities are written in C — a property that gives Unix its amazing portability and accessibility to a wide range of developers.
- Next, Unix has fast process creation time and the unique `fork()` system call. This encourages strongly partitioned systems without gargantuan multi-threaded monstrosities.
- Finally, Unix provides simple yet robust interprocess communication (IPC) primitives that, when coupled with the fast process creation time, allow for the creation of simple utilities that do one thing and do it well, and that can be strung together to accomplish more complicated tasks.

Typical Components of a Kernel

Interrupt handlers: to service interrupt requests
Scheduler: to share processor time among multiple processes
Memory management system: to manage process address spaces
System services: Networking, IPC...

Kernel And Processes

Kernel space

- a protected memory space
- full access to the hardware

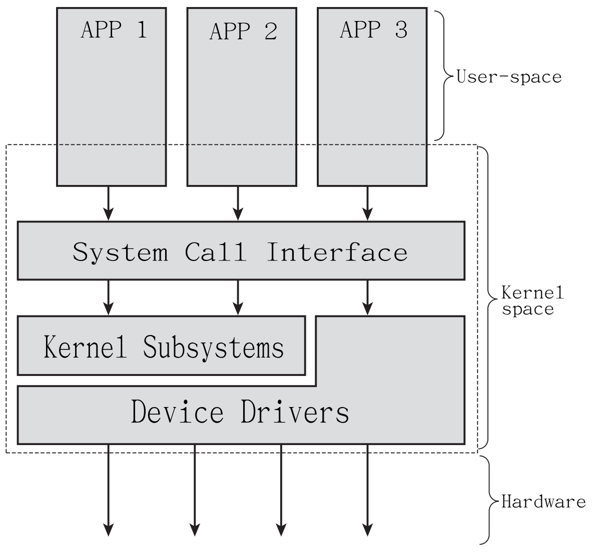
When executing the kernel, the system is in kernel-space executing in *kernel mode*.

User Space

- Can only see a subset of available resources
- unable to perform certain system functions, nor directly access hardware

Normal user execution in user-space executing in *user mode*.

$$\text{user mode} \xrightarrow{system\ calls} \text{kernel mode}$$



C library and system calls An application typically calls functions in a library, for example, the *C library*, that in turn rely on the system call interface to instruct the kernel to carry out tasks on their behalf. Some library calls provide many features not found in the system call, and thus, calling into the kernel is just one step in an otherwise large function. For example, consider the familiar `printf()` function. It provides formatting and buffering of the data and only eventually calls `write()` system call to write the data to the console. Conversely, some library calls have a one-to-one relationship with the kernel. For example, the `open()` library function does nothing except call the `open()` system call. Still other C library functions, such as `strcpy()`, should (you hope) make no use of the kernel at all. When an application executes a system call, it is said that the *kernel is executing on behalf of the application*. Furthermore, the application is said to be *executing a system call in kernel-space*, and the kernel is running in *process context*. This relationship that applications *call into* the kernel via the system call interface is the fundamental manner in which applications get work done. [*Linux Kernel Development*, Chapter 1, *Introduction to the Linux Kernel*]

Kernel And Hardware

Interrupts
Whenever hardware wants to communicate with the system, it issues an interrupt that asynchronously interrupts the kernel.

- Interrupt vector
- Interrupt handlers

Interrupts The kernel also manages the system's hardware. Nearly all architectures, including all systems that Linux supports, provide the concept of *interrupts*. When hardware wants to communicate with the system, it issues an interrupt that asynchronously interrupts the kernel. Interrupts are identified by a number. The kernel uses the number to execute a specific *interrupt handler* to process and respond to the interrupt. For example, as you type, the keyboard controller issues an interrupt to let the system know that there is new data in the keyboard buffer. The kernel notes the interrupt number being issued and executes the correct interrupt handler. The interrupt handler processes the keyboard data and lets the keyboard controller know it is ready for more data. To provide synchronization, the kernel can usually disable interrupts, either all interrupts or just one specific interrupt number. In many operating systems, including Linux, the interrupt handlers do not run in a process context. Instead, they run in a special *interrupt context* that is not associated with any process. This special context exists solely to let an interrupt handler quickly respond to an interrupt, and then exit. [*Linux Kernel Development*, Chapter 1, *Introduction to the Linux Kernel*]

These contexts represent the breadth of the kernel's activities. In fact, in Linux, we can generalize that each processor is doing one of three things at any given moment:

- In kernel-space, in process context, executing on behalf of a specific process
- In kernel-space, in interrupt context, not associated with a process, handling an interrupt
- In user-space, executing user code in a process

This list is inclusive. Even corner cases fit into one of these three activities: For example, when idle, it turns out that the kernel is executing an *idle process* in process context in the kernel.

Kernel Architecture

Monolithic kernels

Simplicity and performance

- exist on disk as single static binaries
- All kernel services run in the kernel address space
- Communication within the kernel is trivial

Most Unix systems are monolithic in design.

Microkernels

- are not implemented as single large processes
- break the kernel into separate processes (*servers*).

  - in the microkernel
    - a few synchronization primitives
    - a simple scheduler
    - an IPC mechanism
  - top of the microkernel
    - memory allocators
    - device drivers
    - system call handlers

Advantages of microkernel OS

- modularized design
- easily ported to other architectures
- make better use of RAM

Performance Overhead

- Communication via *message passing*
- Context switch (kernel-space ⇔ user-space)
  - Windows NT and Mac OS X keep all servers in kernel-space. (defeating the primary purpose of microkernel designs)
- Microkernel OSes are generally slower than monolithic ones.
- Academic research on OS is oriented toward microkernels.

Monolithic Kernel Versus Microkernel Designs Operating kernels can be divided into two main design camps: the monolithic kernel and the microkernel. (A third camp, exokernel, is found primarily in research systems but is gaining ground in real-world use.) [*Linux Kernel Development*, Chapter 1, *Introduction to the Linux Kernel*]

Monolithic kernels involve the simpler design of the two, and all kernels were designed in this manner until the 1980s. Monolithic kernels are implemented entirely as single large processes running entirely in a single address space. Consequently, such kernels typically exist on disk as single static binaries. All kernel services exist and execute in the large kernel address space. Communication within the kernel is trivial because everything runs in kernel mode in the same address space: The kernel can invoke functions directly, as a user-space application might. Proponents of this model cite the simplicity and performance of the monolithic approach. Most Unix systems are monolithic in design.

Microkernels, on the other hand, are not implemented as single large processes. Instead, the functionality of the kernel is broken down into separate processes, usually called servers. Idealistically, only the servers *absolutely* requiring such capabilities run in a privileged execution mode. The rest of the servers run in user-space. All the servers, though, are kept separate and run in different address spaces. Therefore, direct function invocation as in monolithic kernels is not possible. Instead, communication in microkernels is handled via *message passing*: An interprocess communication (IPC) mechanism is built into the system, and the various servers communicate and invoke ”services” from each other by sending messages over the IPC mechanism. The separation of the various servers prevents a failure in one server from bringing down another.

Likewise, the modularity of the system allows one server to be swapped out for another. Because the IPC mechanism involves quite a bit more overhead than a trivial function call, however, and because a context switch from kernel-space to user-space or vice versa may be involved, message passing includes a latency and throughput hit not seen on monolithic kernels with simple function invocation. Consequently, all practical microkernel-based systems now place most or all the servers in kernel-space, to remove the overhead of frequent context switches and potentially allow for direct function invocation. The Windows NT kernel and Mach (on which part of Mac OS X is based) are examples of microkernels. Neither Windows NT nor Mac OS X run any microkernel servers in user-space in their latest versions, defeating the primary purpose of microkernel designs altogether.

Linux is a monolithic kernel, that is, the Linux kernel executes in a single address space entirely in kernel mode. Linux, however, borrows much of the good from microkernels: Linux boasts a modular design with kernel preemption, support for kernel threads, and the capability to dynamically load separate binaries (kernel modules) into the kernel. Conversely, Linux has none of the performance-sapping features that curse micro-kernel designs: Everything runs in kernel mode, with direct function invocation, not message passing, the method of communication. Yet Linux is modular, threaded, and the kernel itself is schedulable. Pragmatism wins again.

1.2 Linux Versus Other Unix-Like Kernels

Linux is a monolithic kernel with modular design
Modularized approach — makes it easy to develop new modules
Platform independence — if standards compliant
Frugal main memory usage — run time (un)loadable
No performance penalty — no explicit message passing is required

Linux
A newcomer in the family of Unix-like OSes (Fig. 1)

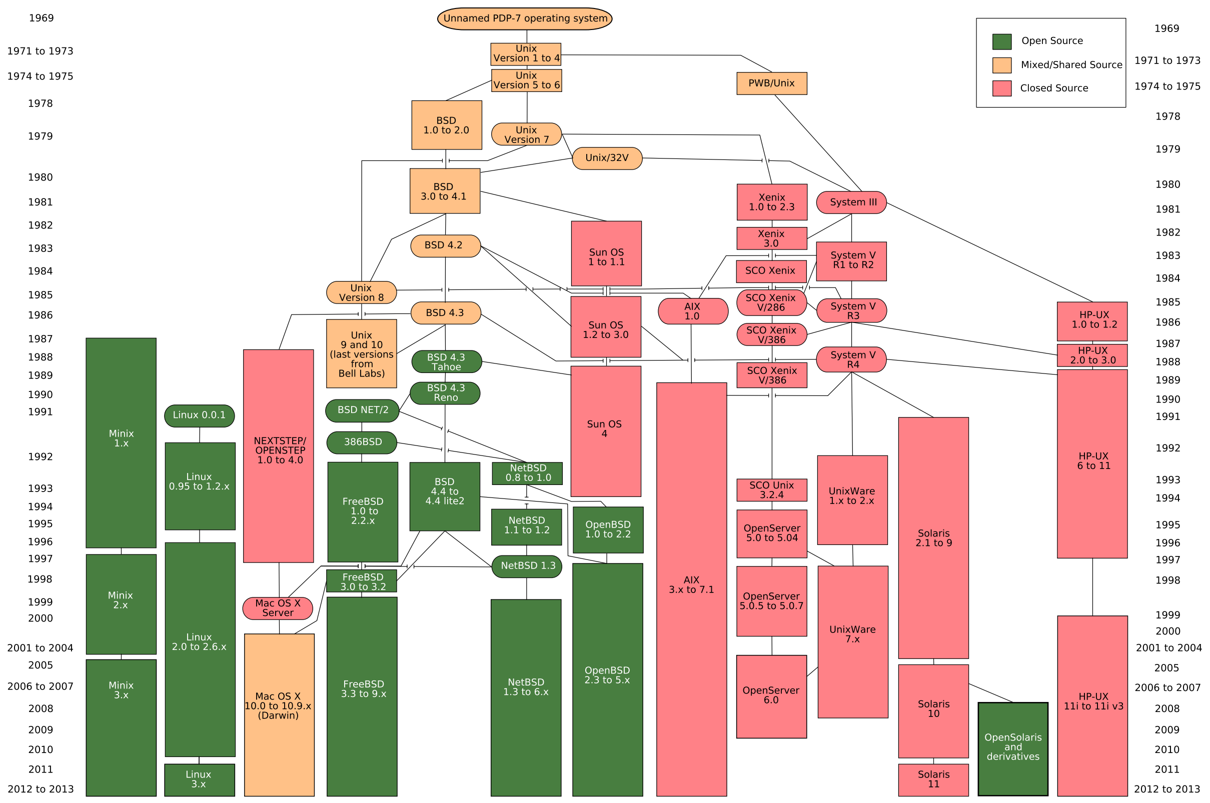


Fig. 1: Unix family tree

Many Unix-like operating systems have arisen over the years, of which Linux is the most popular, having displaced SUS-certified Unix on many server platforms since its inception in the early 1990s. Android, the most widely used mobile operating system in the world, is in turn based on Linux[*Unix — Wikipedia, The Free Encyclopedia*].

Linux Versus Other Unix-Like Kernels

- share fundamental design ideas and features
- from 2.6, Linux kernels are POSIX-compliant
  - Unix programs can be compiled and executed on Linux
- Linux includes all the features of a modern Unix
  - VM, VFS, LWP, SVR4 IPC, signals, SMP support ...

Linux Kernel Features

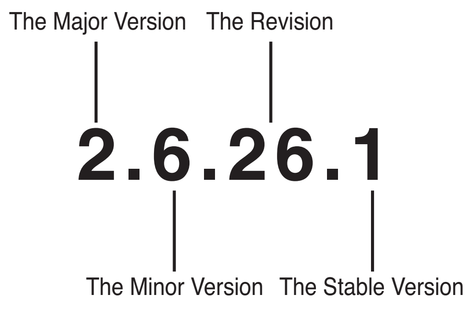
- Monolithic kernel
- loadable modules support
- Kernel threading

- Multithreaded application support
- Preemptive kernel
- Multiprocessor support
- File systems

Advantages Over Its Commercial Competitors

- cost-free
- fully customizable in all its components
- runs on low-end, inexpensive hardware platforms
- performance
- developers are excellent programmers
- kernel can be very small and compact
- highly compatible with many common operating systems
  - filesystems, network interfaces, wine ...
- well supported

Linux Versions



1.3 An Overview of Unix Kernels

1.3.1 The Process/Kernel Model

The Unix Process/Kernel Model

User Mode vs. Kernel Mode

- Processes can run in either user mode or kernel mode
- The kernel itself is not a process but a process manager

$$\text{processes} \xrightarrow{system\ calls} \text{process manager}$$

Kernel routines can be activated in several ways

```
              +-----------+         +-----------+         +-----------+
              |  Process  |         |  Process  |         |  Process  |
              +-----------+         +-----------+         +-----------+
                    |  System             |  Timer              |  Device
   User Mode        |  call               |  Interrupt          |  Interrupt
 -------------------|---------------------|---------------------|-----------
 Kernel Mode        |                     |                     |
                    v                     v                     v
            +--------------+      +--------------+      +--------------+
            | System call  |      |  Scheduler   |      | System call  |
            |   handler    |      |              |      |   handler    |
            +--------------+      +--------------+      +--------------+
```

Unix Kernel Threads

- run in Kernel Mode in the kernel address space
- no interact with users
- created during system startup and remain alive until the system is shut down

1.3.2 Process Implementation

Process Implementation

- Each process is represented by a *process descriptor (PCB)*
- Upon a process switch, the kernel
  - saves the current contents of several registers in the PCB
  - uses the proper PCB fields to load the CPU registers

Registers

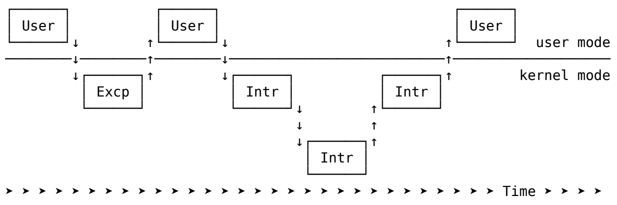
- The program counter (PC) and stack pointer (SP) registers
- The general purpose registers
- The floating point registers
- The processor control registers (Processor Status Word) containing information about the CPU state
- The memory management registers used to keep track of the RAM accessed by the process

1.3.3 Reentrant Kernels

Reentrant Kernels several processes may be executing in Kernel Mode at the same time
    i.e. several processes can wait in kernel mode

Kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.

Interleaving of kernel control paths



Re-entrant functions It's easier to remember when you understand what the term means. The term "re-entrant" means that it is safe to "re-enter" the function while it is already executed, typically in a concurrent environment.

In other words, when two tasks can execute the function at the same time without interfering with each other, then the function is re-entrant. A function is not re-entrant when the execution by one task has an impact on the influence of another task. This typically is the case when a global state or data is used. A function that uses only local variables and arguments is typically re-entrant.

About kernel control path In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves the kernel control paths [*Understanding The Linux Kernel*, Sec 1.6.3, *Reentrant Kernels*]:

case 1 A process executing in user-mode invokes a system call, and the corresponding kernel control path verifies that the request cannot be satisfied immediately; it then invokes the scheduler to select a new process to run. As a result, a process switch occurs. The first kernel control path is left unfinished, and the CPU resumes the execution of some other kernel control path. In this case, the two control paths are executed on behalf of two different processes.

case 2 The CPU detects an exception, for example, access to a page not present in RAM, while running a kernel control path. The first control path is suspended, and the CPU starts the execution of a suitable procedure. In our example, this type of procedure can allocate a new page for the process and read its contents from disk. When the procedure terminates, the first control path can be resumed. In this case, the two control paths are executed on behalf of the same process.

case 3 A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled. The first kernel control path is left unfinished, and the CPU starts processing another kernel control path to handle the interrupt. The first kernel control path resumes when the interrupt handler terminates. In this case, the two kernel control paths run in the execution context of the same process, and the total system CPU time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.

case 4 An interrupt occurs while the CPU is running with kernel preemption enabled, and a higher priority process is runnable. In this case, the first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf of the higher priority process. This occurs only if the kernel has been compiled with kernel preemption support.

See also:

- [*Userspace process preempts kernel thread?*]
- [*Threadsafe vs re-entrant*]
- [*what is the difference between re-entrant function and recursive function in C?*]

1.3.4 Process Address Space

Each process runs in its private address space

- User-mode private stack (user code, data...)
- Kernel-mode private stack (kernel code, data...)

Sharing cases

- The same program is opened by several users
- Shared memory IPC
- `mmap()`

1.3.5 Synchronization and Critical Regions

Re-entrant kernel requires synchronization

If a kernel control path is suspended while acting on a kernel data structure, no other kernel control path should be allowed to act on the same data structure unless it has been reset to a consistent state.

Race condition

When the outcome of a computation depends on how two or more processes are scheduled, the code is incorrect. We say that there is a *race condition*.

- Kernel preemption disabling
- Interrupt disabling
- Semaphores
- Spin locks
- Avoiding deadlocks

1.3.6 Signals and Interprocess Communication

Signals and IPC

Unix signals notifying processes of system events

```
$ man 7 signal
```

IPC semaphores , message queues , and shared memory

```
  shmget(), shmat(), shmdt()
  semget(), semctl(), semop()
  msgget(), msgsnd(), msgrcv()
$ man 5 ipc
```

See also: [*Unix signal — Wikipedia, The Free Encyclopedia*]

1.3.7 Process Management

`fork()` to create a new process
`wait()` to wait until one of its children terminates
`_exit()` to terminate a process
`exec()` to load a new program
`_exit()`: system call; `exit()`: library call

Zombie processes

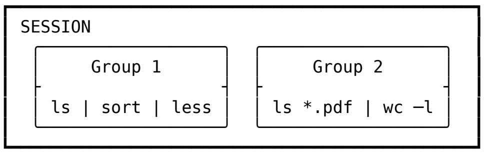
Zombie a *process state* representing terminated processes

- a process remains in that state until its parent process executes a `wait()` system call on it

Orphaned processes become children of *init*.
Read the *NOTES* section of “`man 2 wait`”.

Process groups

```
$ ls | sort | less
$ ls *.pdf | wc -l
```



- `bash` creates a new group for these 3 processes
- each PCB includes a field containing the *process group ID*
- each group of processes may have a *group leader*
- a newly created process is initially inserted into the process group of its parent

login sessions
- All processes in a process group must be in the same login session
- A login session may have several process groups active simultaneously

See also:
- [*Advanced programming in the UNIX environment*, Chapter 9, *Process Relationships*]
- [*What are session leaders in PS?*]
- `man 7 credentials`

1.3.8 Memory Management

Virtual memory

Application memory requests
Virtual memory
MMU

- Several processes can be executed concurrently
- Virtual memory can be larger than physical memory
- Processes can run without fully loaded into physical memory
- Processes can share a single memory image of a library or program
- Easy relocation

RAM Usage

Physical memory
- A few megabytes for storing the kernel image
- The rest of RAM are handled by the virtual memory system
  - dynamic kernel data structures, e.g. buffers, descriptors ...
  - to serve process requests
  - caches of buffered devices

Problems faced:
- the available RAM is limited
- memory fragmentation
- ...

Kernel Memory Allocation

User mode process memory
- Memory pages are allocated from the list of free page frames
- The list is populated using a page-replacement algorithm
- free frames scattered throughout physical memory

Kernel memory allocation
- Treated differently from user memory
  - allocated from a free-memory pool

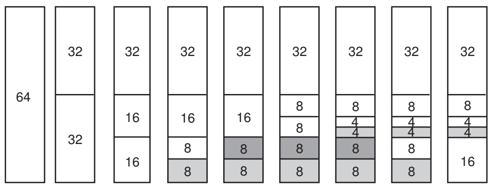
Because:
- must be fast, i.e. avoid searching
- minimize waste, i.e. avoid fragmentation
- maximize contiguousness

Linux's KMA uses a *Slab allocator* on top of a *buddy system*.

See also: [*Operating System Concepts Essentials*, Sec 8.8, *Allocating Kernel Memory*]

Buddy system

- By splitting memory into halves to try to give a best-fit
- Adjacent units of allocatable memory are paired together



Object creation and deletion

- are widely employed by the kernel
- more expensive than allocating memory to them

Slab allocation

- memory chunks suitable to fit data objects of certain type or size are preallocated
  - avoid searching for suitable memory space
  - greatly alleviates memory fragmentation
- Destruction of the object does not free up the memory, but only opens a slot which is put in the list of free slots by the slab allocator

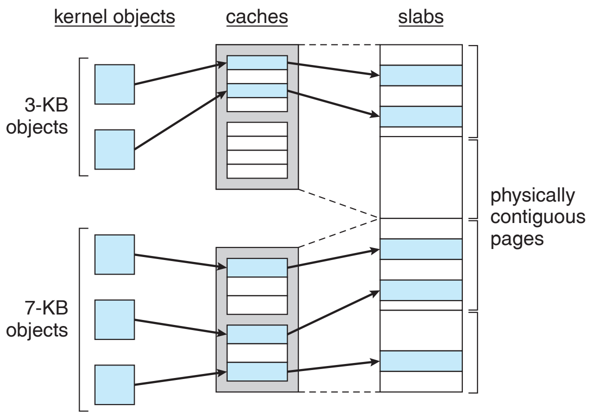
Benefits

- No memory is wasted due to fragmentation
- Memory request can be satisfied quickly

Slab allocation

Slab	is made up of several physically contiguous pages
Cache	consists of one or more slabs.

  - A storage for a specific type of object such as semaphores, process descriptors, file objects etc.



Process virtual address space handling

- demand paging
- copy on write

Caching

- hard drives are very slow
- to defer writing to disk as long as possible
- When a process asks to access a disk, the kernel checks first whether the required data are in the cache
- `sync()`

1.3.9 Device Drivers

The kernel interacts with I/O devices by means of device drivers

The device files in `/dev`

- are the user-visible portion of the device driver interface
- each device file refers to a specific device driver

2 Before Diving Into The Kernel Source

2.1 Getting Started with the Kernel

Ref: [*Linux Kernel Development*, Chapter 2, *Getting Started with the Kernel*]

Obtaining The Kernel Source

Web: `http://www.kernel.org`
Git: Version control system

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

Installing The Kernel Source

In `/usr/src/` directory:

```
$ tar xvjf linux-x.y.z.tar.bz2
$ tar xvzf linux-x.y.z.tar.gz
$ xz -dc linux-x.y.z.tar.xz | tar xf -
$ ln -s linux-x.y.z linux
```

“`sudo`” if needed.

The Kernel Source Tree

arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure

```
$ tree /usr/src/linux
```

Building The Kernel

Configuration

```
make config | make menuconfig | make gconfig
```

```
$ make defconfig # Defaults
$ make oldconfig # Using .config file
```

Make

```
$ make -jN > /dev/null
```

N: number of jobs. Usually one or two jobs per processor.

Installing The New Kernel

```
$ sudo make modules_install
$ sudo make install
```

`grub2` config should be updated automatically. Check

- `/etc/grub.d/`
- `/boot/grub/grub.cfg`

"`sudo reboot`" to try your luck.

See also: [*Debian Linux Kernel Handbook*]

2.2 A Beast Of A Different Nature

- Neither the C library nor the standard C headers
- GNU C
- Lack of memory protection
- No floating-point operations
- Small per-process fixed-size stack
- Synchronization and concurrency
- Portability

No libc or standard headers

- Chicken-and-the-egg situation
- Speed and size

Many of the usual libc functions are implemented inside the kernel. For example,

- String operation: `lib/string.c`, `linux/string.h`
- `printk()`

GNU C

The kernel developers use both `ISO C99` and `GNU C` extensions to the C language.

- Inline functions
- Inline assembly

Inline functions

Inserted inline into each function call site

- ☺ eliminates the overhead of function invocation and return (register saving and restore)
- ☺ allows for potentially greater optimization
- ☹ code size increases

```
static inline void wolf(unsigned long tail_size)
```

- Kernel developers use inline functions for small time-critical functions
- The function declaration must precede any usage
  Common practice: place inline functions in header files

See also:

- [*Inline function — Wikipedia, The Free Encyclopedia*]
- [*Inline Functions In C*]

Inline assembly

Embedding assembly instructions in normal C functions

- ☺ speed
- ☹ Architecture dependent (poor potability)

Example: Get the value from the timestamp(tsc) register

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=b" (high));
```

See also: [*Inline assembler — Wikipedia, The Free Encyclopedia*]

Branch prediction

The `likely()` and `unlikely()` macros allow the developer to tell the CPU, through the compiler, that certain sections of code are likely, and thus should be predicted, or unlikely, so they shouldn't be predicted.

```
#define   likely(x)  __builtin_expect(!!(x),1)
#define unlikely(x)  __builtin_expect(!!(x),0)
```

Example: `kernel/time.c`

```
asmlinkage long sys_gettimeofday(struct timeval __user *tv, struct timezone __user *tz)
{
        if (likely(tv != NULL)) {
                struct timeval ktv;
                do_gettimeofday(&ktv);
                if (copy_to_user(tv, &ktv, sizeof(ktv)))
                        return -EFAULT;
        }
        if (unlikely(tz != NULL)) {
                if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
                        return -EFAULT;
        }
        return 0;
}
```

In this code, we see that a syscall to get the time of day is likely to have a `timeval` structure that is not null. If it were null, we couldn't fill in the requested time of day! It is also unlikely that the timezone is not null. To put it another way, the caller rarely asks for the timezone and usually asks for the time. [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Sec. 2.8.2, *`likely()` and `unlikely()`*]

`!!(x)` make sure `x` is a boolean variable, since in C, macro invocations do not perform type checking, or even check that arguments are well-formed.

See also:

- [*FAQ/LikelyUnlikely*]
- [*Inline function — Wikipedia, The Free Encyclopedia*]

No memory protection

- Memory violations in the kernel result in an *oops*
- Kernel memory is not pageable

See also: [*Linux kernel oops — Wikipedia, The Free Encyclopedia*].

No (easy) use of floating point

- rarely needed
- expensive: saving the FPU registers and other FPU state takes time
- not every architecture has a FPU, e.g. those for embedded systems

See also:

- [*Use of floating point in the Linux kernel*]
- [*Linux Kernel and Floating Point*]

Small, fixed-size stack

- On x86, the stack size is configurable at compile time, 4K or 8K (1 or 2 pages)

2.3 Common Kernel Data-types

- [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Chapter 2, *Exploration Toolkit*];
- [*Linux Kernel Development*, Chapter 6, *Kernel Data Structures*];
- [*Linux Device Drivers*, Chapter 11, *Data Types in the Kernel*]

Data Types in the Kernel

Three main classes:

1. Standard C types, e.g. `int`
2. Explicitly sized types, e.g. `u32`
3. Types used for special kernel objects, e.g. `pid_t`

Although you must be careful when mixing different data types, sometimes there are good reasons to do so. One such situation is for memory addresses, which are special as far as the kernel is concerned. Although, conceptually, addresses are pointers, *memory administration is often better accomplished by using an unsigned integer type; the kernel treats physical memory like a huge array, and a memory address is just an index into the array. Furthermore, a pointer is easily dereferenced; when dealing directly with memory addresses, you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs. Therefore, generic memory addresses in the kernel are usually `unsigned long`, exploiting the fact that pointers and long integers are always the same size, at least on all the platforms currently supported by Linux.* [*Linux Device Drivers*, p289]

Common Kernel Data-types

Many objects and structures in the kernel

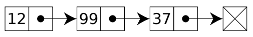
- memory pages
- processes
- interrupts
- ...

1. linked-lists — to group them together
2. binary search trees — to efficiently find a single element

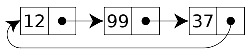
2.3.1 Linked Lists

- [*Linux Device Drivers*, Chapter 11, *Data Types in the Kernel*]
- [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Sec. 2.1, *Common Kernel Datatypes*]

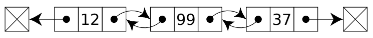
Singly linked list



Circularly linked list



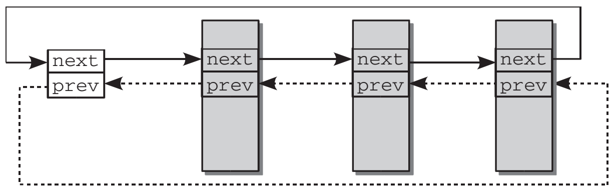
Doubly linked list



Linked lists are ill-suited for use cases where *random access* is an important operation. Instead, you use linked lists when *iterating over the whole list* is important and the *dynamic addition and removal* of elements is required. [*Linux Kernel Development*, Sec. 6.1.3, *Moving Through a Linked List*]

See also: [*Array versus linked-list*].

Circular doubly linked-lists



See also:

- [*Understanding The Linux Kernel*, Sec. 3.2.2.3, *Doubly Linked Lists*];
- [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Sec. 2.1, *Common Kernel Data Types*];
- [*FAQ/LinkedLists*];
- [*Linux Kernel Linked List Explained*].

Things to note

1. List is *inside the data item* you want to link together.
2. You **can put** "`struct list_head`" **anywhere** in your structure.
3. You **can name** "`struct list_head`" **variable anything** you wish.
4. You **can have** multiple lists!

Linked Lists

A linked list is initialized by using the `LIST_HEAD` and `INIT_LIST_HEAD` macros

`include/linux/list.h`

```
struct list_head {
        struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
        (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

Q1: Why both `LIST_HEAD_INIT` and `INIT_LIST_HEAD`?
Q2: Why `do-while`?

LIST_HEAD When first encountering this, most people are confused because they have been taught to implement linked lists by adding a pointer in a structure which points to the next similar structure in the linked list. The drawback of this approach, and the reason for which the kernel implements linked lists differently, is that you need to write code to handle (adding/removing/etc) elements specifically for that data structure. Here, we can add a struct `list_head` field to any other data structure and, as we'll see shortly, make it a part of a linked list. Moreover, if you want your data structure to be part of several data structures, adding a few of these fields will work out just fine[1]. [*FAQ/LinkedLists*]

Why is there a need to declare `LIST_HEAD_INIT` and `INIT_LIST_HEAD` they both seem to do the same thing?

- An example [*Linux Kernel Development*, Sec. 6.1.1, *Defining a linked list*].
- `LIST_HEAD_INIT` initializes one of these list head structure thingamajigs at compile time and so it doesn't actually generate any code, but stores constant values in a structure of which the address is known and fixed[2]. [*link list implementation in linux kernel*]
  On the other hand, `INIT_LIST_HEAD` takes a pointer and so it could be a structure that is dynamically allocated, e.g. with "`malloc`" or one that is passed into your function... and then it does it's stuff at run time.
- `LIST_HEAD_INIT` is a static initializer, `INIT_LIST_HEAD` is a function (in 2.6.11 it's still a macro). They both initialise a `list_head` to be empty [3]. [*LIST_HEAD_INIT vs INIT_LIST_HEAD*]
  If you are statically declaring a `list_head`, you should use `LIST_HEAD_INIT`, e.g.:

  ```
  static struct list_head mylist = INIT_LIST_HEAD(mylist);
  ```

  You should use `INIT_LIST_HEAD()` for a list head that is dynamically allocated, usually part of another structure. There are many examples in the kernel source.
- **My understanding:** compile time list head initialization is better for kernel memory management e.g. slab allocation.
- See also: [*C dynamic memory allocation — Wikipedia, The Free Encyclopedia*]

Example

```
struct fox {
  unsigned long tail_length;
  unsigned long weight;
  bool is_fantastic;
  struct list_head list;
};
```

The list needs to be initialized before in use

- Run-time initialization

```
struct fox *red_fox; /* just a pointer */
red_fox = kmalloc(sizeof(*red_fox), GFP_KERNEL);
red_fox->tail_length = 40;
red_fox->weight = 6;
red_fox->is_fantastic = false;
INIT_LIST_HEAD(&red_fox->list);
```

[1] http://kernelnewbies.org/FAQ/LinkedLists
[2] http://ubuntuforums.org/archive/index.php/t-1591281.html
[3] http://stackoverflow.com/questions/10262017/linux-kernel-list-list-head-init-vs-init-list-head

- Compile-time initialization

```
struct fox red_fox = {
  .tail_length = 40,
  .weight = 6,
  .list = LIST_HEAD_INIT(red_fox.list),
};
```

See also:
- [*Linux Kernel Development*, Sec. 6.1.4, *The Linux Kernel's Implementation*]
- About `kmalloc()`, see [*Linux Device Drivers*, Sec. 8.1, *The Real Story of kmalloc*]
- `GFP MASK` [*PageAllocation*]

The `do while(0)` trick

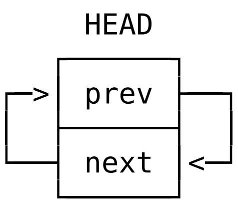
```
#define INIT_LIST_HEAD(ptr) do {                      \
     (ptr)->next = (ptr); (ptr)->prev = (ptr);        \
  } while (0)

if (1)
  INIT_LIST_HEAD(x);
 else
   error(x);

/* after "gcc -E macro.c" */
if (1)
  do { (x)->next = (x); (x)->prev = (x); } while (0);
 else
   error(x);

/********************* Wrong ********************/
#define INIT_LIST_HEAD2(ptr) {                        \
     (ptr)->next = (ptr); (ptr)->prev = (ptr);        \
  }

if (1)
  INIT_LIST_HEAD2(x); /* the semicolon is wrong! */
 else
   error(x);

/* after "gcc -E macro.c" */
if (1)
  { (x)->next = (x); (x)->prev = (x); };
 else
   error(x);
```

do...while(0)

```
/* Note: they use dummy do {} while construct so that
   it can be used syntactically as a single statement. */
#define INIT_LIST_HEAD(ptr) do {              \
    (ptr)->next = (ptr);(ptr)->prev= (ptr);   \
} while(0)
```

See also:
- [*link list implementation in linux kernel easy question syntax not clear*]
- [*What's the use of do while(0) when we define a macro?*]
- [*Do-While and if-else statements in C/C++ macros*]

After `INIT_LIST_HEAD` macro is called



```
struct list_head {
        struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
        (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

Example: to start an empty `fox` list

```
static LIST_HEAD(fox_list);
```

Manipulating linked lists

- To add a new member into `fox_list`:

```
list_add(&new->list,&fox_list);
  * fox_list->next->prev = new->list;
  * new->list->next = fox_list->next;
  * new->list->prev = fox_list;
  * fox_list->next = new->list;
list_add_tail(&f->list,&fox_list);
```

- To remove an `old` node from list:

```
list_del(&old->list);
  * old->list->next->prev = old->list->prev;
  * old->list->prev->next = old->list->next;
```

- a lot more...

This function adds the new node to the given list immediately after the head node.

```
list_add(struct list_head *new, struct list_head *head)
```

Because the list is circular and generally has no concept of *first* or *last* nodes, you can pass any element for `head`. If you do pass the "`last`" element, however, this function can be used to implement a stack [*Linux Kernel Development*, Sec. 6.1.5, *Manipulating Linked Lists*].

List Traversing

```
struct list_head *p;
list_for_each(p,fox_list){ ... }
```

`list_for_each`

```
/**
 * list_for_each       -       iterate over a list
 * @pos:        the &struct list_head to use as a loop counter.
 * @head:       the head for your list.
 */
#define list_for_each(pos, head) \
        for (pos = (head)->next; prefetch(pos->next), pos != (head); \
                pos = pos->next)
```

Not so useful. Usually we want the pointer to the container struct.

```
struct fox {
  unsigned long tail_length;
  unsigned long weight;
  bool is_fantastic;
  struct list_head list;
};
```

Q: Given a pointer to *list*, how to get a pointer to *fox*?

```
f = list_entry(p, struct fox, list);
```

`list_entry(ptr, type, member)`

```
/**
 * list_entry - get the struct for this entry
 * @ptr:        the &struct list_head pointer.
 * @type:       the type of the struct this is embedded in.
 * @member:     the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)

#define container_of(ptr, type, member) ({                    \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);  \
        (type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

The code above deserves some explanation (Also read [*FAQ/ContainerOf*]).

- `container_of(ptr,type,member)`: We have the address of the `struct list_head` field inside of another data structure (let's say `struct task_struct` for sake of example). The first line of the macro casts the value `0` into a pointer of the encapsulating data structure type (`struct task_struct` in our example). We use this pointer to access the field in that data structure which corresponds to our `list_head` and get its type with the macro `typeof` to declare a pointer `mptr` initialized to the value contained in `ptr`. [*FAQ/LinkedLists*]
- How Does This Work? See [*Linux Kernel Linked List Explained*]
- `list_entry(ptr,type,member)`: [*Linux Kernel: $\boxed{\text{F}}$大又好用的 list_head 結構*]

`list_for_each_entry(pos, head, member)`

```
struct fox *f;

list_for_each_entry(f, &fox_list, list) {
  /* on each iteration, 'f' points to the next fox structure ... */
}
```

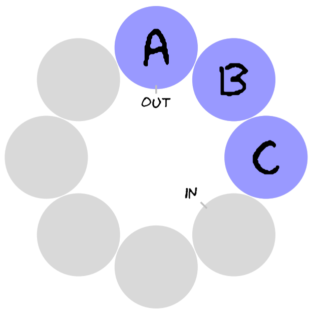
`list_for_each_entry(pos, head, member)`

```
/**
 * list_for_each_entry  -       iterate over list of given type
 * @pos:        the type * to use as a loop counter.
 * @head:       the head for your list.
 * @member:     the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member)                        \
        for (pos = list_entry((head)->next, typeof(*pos), member);    \
             prefetch(pos->member.next), &pos->member != (head);      \
             pos = list_entry(pos->member.next, typeof(*pos), member))
```

2.3.2 Queues

Ref: [*Linux Kernel Development*, Sec 6.2, *Queues*]

Queue (FIFO)



Circular buffer
Empty: $in == out$
Full: $(in + 1)\%BUFFER\_SIZE == out$

Can be lock-free.

See also: [*Circular buffer — Wikipedia, The Free Encyclopedia*]

KFIFO

`include/linux/kfifo.h`

```
struct kfifo {
        unsigned char *buffer;   /* the buffer holding the data */
        unsigned int size;       /* the size of the allocated buffer */
        unsigned int in;         /* data is added at offset (in % size) */
        unsigned int out;        /* data is extracted from off. (out % size) */
        spinlock_t *lock;        /* protects concurrent modifications */
};
```

The *spinlock* is rarely needed.
Initialization:

```
int kfifo_alloc(struct kfifo *fifo,
                unsigned int size,
                gfp_t gfp_mask);
void kfifo_init(struct kfifo *fifo,
                void *buffer,
                unsigned int size);
```

Example: to have a `PAGE_SIZE`-sized queue

```
struct kfifo fifo;
int ret;
ret = kfifo_alloc(&fifo, PAGE_SIZE, GFP_KERNEL);
if (ret)
  return ret;
```

See also: [*Linux Kernel Development*, Sec. 6.2.2, *Creating a queue*, p97]

`kfifo` operations

```
/* Enqueue */
unsigned int kfifo_in(struct kfifo *fifo,
                      const void *from, unsigned int len);
/* Dequeue */
unsigned int kfifo_out(struct kfifo *fifo,
                       void *to, unsigned int len);
/* Peek */
unsigned int kfifo_out_peek(struct kfifo *fifo, void *to,
                            unsigned int len, unsigned offset);
/* Get size */
static inline unsigned int kfifo_size(struct kfifo *fifo);

/* Get queue length */
static inline unsigned int kfifo_len(struct kfifo *fifo);

/* Get available space */
static inline unsigned int kfifo_avail(struct kfifo *fifo);

/* Is it empty? */
static inline int kfifo_is_empty(struct kfifo *fifo);

/* Is it full? */
static inline int kfifo_is_full(struct kfifo *fifo);

/* Reset */
static inline void kfifo_reset(struct kfifo *fifo);

/* Destroy (kfifo_alloc()ed only) */
void kfifo_free(struct kfifo *fifo);
```

Example

1. To enqueue 32 integers into *fifo*

```
unsigned int i;
for (i = 0; i < 32; i++)
  kfifo_in(fifo, &i; sizeof(i));
```

2. To dequeue and print all the items in the queue

```
/* while there is data in the queue ... */
while (kfifo_len(fifo)) {
  unsigned int val;
  int ret;
  /* ... read it, one integer at a time */
  ret = kfifo_out(fifo, &val, sizeof(val));
  if (ret != sizeof(val))
    return -EINVAL;
  printk(KERN_INFO "%u\n", val);
 }
```

- `kfifo_in()`: This function copies the *len* bytes starting at *from* into the queue represented by *fifo*. [*Linux Kernel Development*, Sec. 6.2.3 *Enqueuing Data*, p98]
  - On success it returns the number of bytes enqueued.
  - If less than *len* bytes are free in the queue, the function copies only up to the amount of available bytes.Thus the return value can be less than *len* or even zero, if nothing was copied.
- `kfifo_out()`: This function copies at most *len* bytes from the queue pointed at by *fifo* to the buffer pointed at by *to*. [*Linux Kernel Development*, Sec. 6.2.4, *Dequeuing Data*, p98]
  - On success the function returns the number of bytes copied.
  - If less than *len* bytes are in the queue, the function copies less than requested.

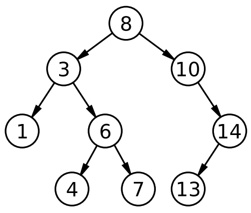
2.3.3 Binary Trees

Ref: [*Linux Kernel Development*, Sec. 6.4, *Binary Trees*]

Trees

- Used in Linux memory management
  - fast store/retrieve a single piece of data among many
- generally implemented as *linked lists* or *arrays*
- the process of moving through a tree — *traversing*

Binary Search Tree



Properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Efficient in:

1. searching for a given node
2. in-order traversal (e.g. Left-Root-Right)

See also:

- [*Binary search tree — Wikipedia, The Free Encyclopedia*]
- `http://www.cs.duke.edu/~reif/courses/alglectures/skiena.lectures/lecture8.pdf`
- `http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/bst.pdf/`
- `http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/binarySearchTree.htm`

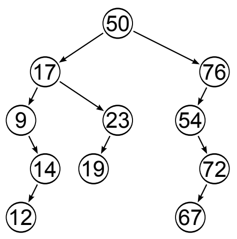


Fig. 2: Unbalanced binary tree

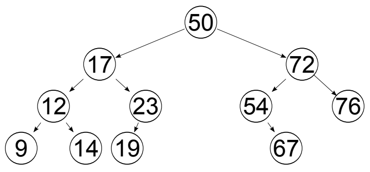


Fig. 3: Balanced binary tree

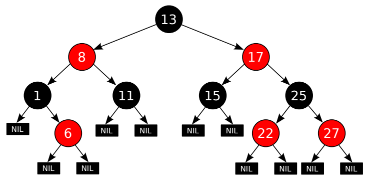


Fig. 4: Red-black tree

- A *balanced binary search tree* is a binary search tree in which the depth of all leaves differs by at most one.
- A *self-balancing binary search tree* is a binary search tree that attempts, as part of its normal operations,to remain (semi) balanced.

See also:

- [*Tree (data structure) — Wikipedia, The Free Encyclopedia*]
- [*Binary tree — Wikipedia, The Free Encyclopedia*]
- [*Binary search tree — Wikipedia, The Free Encyclopedia*]
- [*AVL tree — Wikipedia, The Free Encyclopedia*]
- [*Red–black tree — Wikipedia, The Free Encyclopedia*]
- [*Big O notation — Wikipedia, The Free Encyclopedia*]

Red-black tree A type of *self-balancing BST* in which each node has a red or black color attribute.

Properties to make it *semi-balanced*:

1. All nodes are either red or black
2. Leaf nodes are black (root's color)
3. Leaf nodes do not contain data (NULL)
4. All non-leaf nodes have two children
5. If a node is red, both its children are black
6. When traversing from the root node to a leaf, each path contains the same number of black nodes

These properties ensure that *the deepest leaf has a depth of no more than double that of the shallowest leaf.*

- Property 5 implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
- Taken together, these properties ensure that the deepest leaf has a depth of no more than double that of the shallowest leaf. Consequently, the tree is always semi-balanced. Why this is true is surprisingly simple. First, by property five, a red node cannot be the child or parent of another red node. By property six, all paths through the tree to its leaves have the same number of black nodes.The longest path through the tree alternates red and black nodes.Thus the shortest path, which must have the same number of black nodes, contains only

black nodes.Therefore, the longest path from the root to a leaf is no more than double the shortest path from the root to any other leaf. [*Linux Kernel Development*, p105]
- *Data structure and algorithms, Sec 8.2, Red-black tree*[4]
- Red-black tree Java applet demo [5]

Advantages
- faster real-time bounded worst case performance for insertion and deletion
  - usually at most two rotations
- slightly slower (but still $O(logn)$) lookup time

Many red-black trees in use in the kernel
- The deadline and CFQ I/O schedulers employ rbtrees to track requests;
- the packet CD/DVD driver does the same
- The high-resolution timer code uses an rbtree to organize outstanding timer requests
- The ext3 filesystem tracks directory entries in a red-black tree
- Virtual memory areas (VMAs) are tracked with red-black trees
- epoll file descriptors, cryptographic keys, and network packets in the "hierarchical token bucket" scheduler

See also:
- [*Red-black trees*]
- [*Red-black Trees (rbtree) in Linux*]

`<linux/rbtree.h>`

```
struct rb_node
{
        struct rb_node *rb_parent;
        int rb_color;
#define RB_RED          0
#define RB_BLACK        1
        struct rb_node *rb_right;
        struct rb_node *rb_left;
};

struct rb_root
{
        struct rb_node *rb_node;
};

#define RB_ROOT (struct rb_root) { NULL, }
#define rb_entry(ptr, type, member)       \
        container_of(ptr, type, member)
```

To create a new empty tree:

```
struct rb_root root = RB_ROOT;
```

- Linux 内核中的红黑树[6]
- "`(struct rb_root){ NULL, }`" is a compound literal [*GCC Manual*, Sec. 6.26]. "`{ NULL, }`" is an *initializer list* [*ISO C Standard 1999*, Sec. 6.7.8, p125, *Initialization*]. It initializes the 1st element (a pointer) to *NULL*, and initializes the rest (omitted) as if they are static objects: arithmetic types are initialized to *0*; pointers are initialized to *NULL* [*Designated initializers for aggregate types (C only)*].
  See also:
  - [*ISO C Standard 1999*, Sec. 6.5.2.5, p75, *Compound literals*]
  - [*ISO C Standard 1999*, Sec. 6.27, *Designated Initializers*]

Example

```
struct fox {
  struct rb_node node;
  unsigned long tail_length;
  unsigned long weight;
  bool is_fantastic;
};
```

Search

[4] http://www.cs.auckland.ac.nz/~jmor159/PLDS210/red_black.html
[5] http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html
[6] http://www.kerneltravel.net/jiaoliu/kern-rbtree.html

```
struct fox *fox_search(struct rb_root *root, unsigned long ideal_length)
{
  struct rb_node *node = root->rb_node;

  while (node) {
    struct fox *a_fox = container_of(node, struct fox, node);

    int result;

    result = tail_compare(ideal_length, a_fox->tail_length);

    if (result < 0)
      node = node->rb_left;
    else if (result > 0)
      node = node->rb_right;
    else
      return a_fox;
  }
  return NULL;
}
```

Example

Searching for a specific page in the page cache

```
struct page *rb_search_page_cache(struct inode *inode,
                                  unsigned long offset)
{
  struct rb_node *n = inode->i_rb_page_cache.rb_node;
  while (n) {
    struct page *page = rb_entry(n, struct page, rb_page_cache);
    if (offset < page->offset)
      n = n->rb_left;
    else if (offset > page->offset)
      n = n->rb_right;
    else
      return page;
  }
  return NULL;
}
```

See also:
- `include/linux/rbtree.h`
- [*Linux Kernel Development*, p106-107]
- [*Understanding The Linux Kernel*, Sec. 9.3, *Memory Regions*]

2.4 Assembly

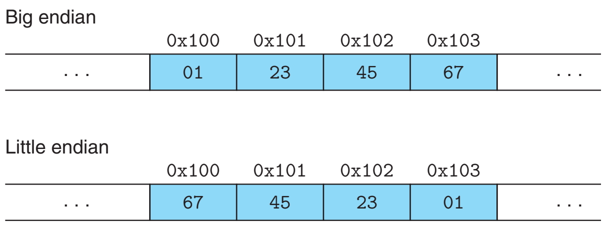
x86 Assembly

The Pentium class x86 architecture

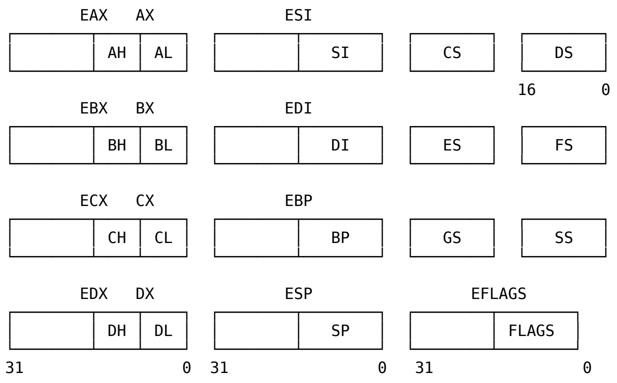
- Data ordering is in Little Endian
- Memory access is in byte (8 bit), word (16 bit), double word (32 bit), and quad word (64 bit).
- the usual registers for code and data instructions can be broken down into three categories: *control*, *arithmetic*, and *data*.

Byte ordering is architecture dependent

Storing an int (`0x01234567`) at address `0x100`:



See also: [*Endianness — Wikipedia, The Free Encyclopedia*]



Three kinds of registers

1. general purpose registers
2. segment registers
3. status/control registers

General purpose registers

EAX	Accumulator register	**ESI**	Source Index
EBX	Base register	**EDI**	Destination Index
ECX	Counter for loop operations	**ESP**	Stack Pointer
EDX	Data register	**EBP**	Base Pointer pointing to the top of previous stack frame

The eight general-purpose registers in the x86 processor family each have an unique purpose. Each register has special instructions and opcodes which make fulfilling this purpose more convenient or efficient. [*The Art of Picking Intel Registers*]

EAX There are three major processor architectures: register, stack, and accumulator. In a register architecture, operations such as addition or subtraction can occur between any two arbitrary registers. In a stack architecture, operations occur between the top of the stack and other items on the stack. In an accumulator architecture, the processor has single calculation register called the accumulator. All calculations occur in the accumulator, and the other registers act as simple data storage locations.

The x86 processor does not have an accumulator architecture. It does, however, have an accumulator-like register: `EAX/AL`. Although most calculations can occur between any two registers, the instruction set gives the accumulator special preference as a calculation register.

Since most calculations occur in the accumulator, the x86 architecture contains many optimized instructions for moving data in and out of this register. ... In your code, try to perform as much work in the accumulator as possible. As you will see, the remaining seven general-purpose registers exist primarily to support the calculation occurring in the accumulator.

EBX The base register gets its name from the `XLAT` instruction. `XLAT` looks up a value in a table using `AL` as the index and `EBX` as the base. `XLAT` is equivalent to `MOV AL, [BX+AL]`, which is sometimes useful if you need to replace one 8-bit value with another from a table (Think of color look-up).

So, of all the general-purpose registers, `EBX` is the only register without an important dedicated purpose. It is a good place to store an extra pointer or calculation step, but not much more.

EDX Of the seven remaining general-purpose registers, the data register, `EDX`, is most closely tied to the accumulator. Instructions that deal with over sized data items, such as multiplication, division, `CWD`, and `CDQ`, store the most significant bits in the data register and the least significant bits in the accumulator. In a sense, the data register is the 64-bit extension of the accumulator. The data register also plays a part in I/O instructions. In this case, the accumulator holds the data to read or write from the port, and the data register holds the port address.

ECX The count register, `ECX`, is the x86 equivalent of the ubiquitous variable `i`. Every counting-related instruction in the x86 uses `ECX`. The most obvious counting instructions are `LOOP`, `LOOPZ`, and `LOOPNZ`. Another counter-based instruction is `JCXZ`, which, as the name implies, jumps when the counter is 0. The count register also appears in some bit-shift operations, where it holds the number of shifts to perform. Finally, the count register controls the string instructions through the `REP`, `REPE`, and `REPNE` prefixes. In this case, the count register determines the maximum number of times the operation will repeat.

Particularly in demos, most calculations occur in a loop. In these situations, `ECX` is the logical choice for the loop counter, since no other register has so many branching operations built around it. The only problem is that this register counts downward instead of up as in high level languages. Designing a downward-counting is not hard, however, so this is only a minor difficulty.

EDI Every loop that generates data must store the result in memory, and doing so requires a moving pointer. The destination index, EDI, is that pointer. The destination index holds the implied write address of all string operations. The most useful string instruction, remarkably enough, is the seldom-used `STOS`. `STOS` copies data from the accumulator into memory and increments the destination index. This one-byte instruction is perfect, since the final result of any calculation should be in the accumulator anyhow, and storing results in a moving memory address is a common task.
See also: [*What does `rep stos` do?*]

ESI The source index, `ESI`, has the same properties as the destination index. The only difference is that the source index is for reading instead of writing. Although all data-processing routines write, not all read, so the source index is not as universally useful. When the time comes to use it, however, the source index is just as powerful as the destination index, and has the same type of instructions.
In situations where your code does not read any sort of data, of course, using the source index for convenient storage space is acceptable.

ESP & EBP When a block of code calls a function, it pushes the parameters and the return address on the stack. Once inside, function sets the base pointer equal to the stack pointer and then places its own internal variables on the stack. From that point on, the function refers to its parameters and variables relative to the base pointer rather than the stack pointer. Why not the stack pointer? For some reason, the stack pointer lousy addressing modes. In 16-bit mode, it cannot be a square-bracket memory offset at all. In 32-bit mode, it can be appear in square brackets only by adding an expensive SIB byte to the opcode.
In your code, there is never a reason to use the stack pointer for anything other than the stack. The base pointer, however, is up for grabs. If your routines pass parameters by register instead of by stack (they should), there is no reason to copy the stack pointer into the base pointer. The base pointer becomes a free register for whatever you need.

Segment registers

CS Code segment
SS Stack segment
DS,ES,FS,GS Data segment

A memory address is an offset in a segment

ES:EDI references memory in the ES (extra segment) with an offset of the value in the EDI
DS:ESI
CS:EIP
SS:ESP

State/Control registers

EFLAGS Status, control, and system flags
EIP The instruction pointer, contains an offset from CS (CS:EIP)

FLAGS

15				11				7	6						0
-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C

CF Carry flag
ZF Zero flag
SF Sign flag, Negative flag
OF Overflow flag

See also: [*FLAGS register — Wikipedia, The Free Encyclopedia*]

Control Instructions (Intel syntax)

Instruction	Function	EFLAGS
`je`	`Jump if equal`	$ZF = 1$
`jg`	`Jump if greater`	$ZF = 0$
		$SF = OF$
`jge`	`Jump if greater or equal`	$SF = OF$
`jl`	`Jump if less`	$SF \neq OF$
`jle`	`Jump if less or equal`	$ZF = 1$
`jmp`	`Unconditional jump`	`unconditional`

Example (Intel syntax)

```
  pop eax       ; Pop top of the stack into eax
loop2:
  pop ebx
  cmp eax, ebx ; Compare the values in eax and ebx
  jge loop2     ; Jump if eax >= ebx
```

Data can be moved

- between registers
- between registers and memory
- from a constant to a register or memory, but
- **NOT** from one memory location to another

Data instructions (Intel syntax)

1. `mov eax, ebx`
   Move 32 bits of data from `ebx` to `eax`
2. `mov eax, WORD PTR[data3]`
   Move 32 bits of data from memory variable `data3` to `eax`
3. `mov BYTE PTR[char1], al`
   Move 8 bits of data from `al` to memory variable `char1`
4. `mov eax, 0xbeef`
   Move the constant value `0xbeef` to `eax`
5. `mov WORD PTR[my_data], 0xbeef`
   Move the constant value `0xbeef` to the memory variable `my_data`

Address operand syntax (AT&T syntax)

```
ADDRESS_OR_OFFSET(%BASE_OR_OFFSET, %INDEX, MULTIPLIER)
```

- up to 4 parameters
  - `ADDRESS_OR_OFFSET` and `MULTIPLIER` must be **constants**
  - `%BASE_OR_OFFSET` and `%INDEX` must be **registers**
- all of the fields are optional
  - if any of the pieces is left out, substituted it with zero
- final address =
  `ADDRESS_OR_OFFSET + %BASE_OR_OFFSET + %INDEX * MULTIPLIER`

Why so complicate?
To serve several *addressing modes*
direct addressing mode `movl ADDRESS, %eax`

- load data at `ADDRESS` into `%eax`

indexed addressing mode `movl START(,%ecx,1), %eax`

- `START` – starting address; - `%ecx` – offset/index

indirect addressing mode `movl (%eax), %ebx`

- load data at address pointed by `%eax` into `%ebx`
- `%eax` contents an address pointer

base pointer addressing mode `movl 4(%eax), %ebx`

immediate mode `movl $12, %eax`
without $ Direct addressing

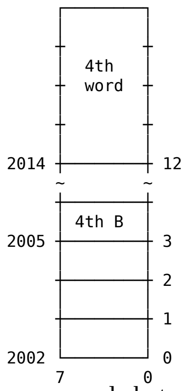
indexed addressing mode

```
movl START(,%ecx,1), %eax
```

`START` starting address
`%ecx` offset/index

`START(,INDEX,MULTIPLIER):`

- to access the 4th byte from location 2002
  $2002(,3,1) = 2002 + 3 \times 1 = 2005$
- to access the 4th word from location 2002
  $2002(,3,4) = 2002 + 3 \times 4 = 2014$



In *indexed addressing mode*, if you have a set of numbers starting at location 2002, you can cycle between each of them using an index register.

The *multiplier* allows you to access memory a byte at a time or a word at a time (4 bytes).

e.g. If you are accessing an entire word, your index will need to be multiplied by 4 to get the exact location of the fourth element from your address.
For example, if you wanted to access the fourth byte from location 2002, you would load your index register with 3 (remember, we start counting at 0) and set the multiplier to 1 since you are going a byte at a time. This would get you location 2005. However, if you wanted to access the fourth word from location 2002, you would load your index register with 3 and set the multiplier to 4. This would load from location 2014 - the fourth word. Take the time to calculate these yourself to make sure you understand how it works.

See also: [*Programming from the Ground Up*, Sec. 2.5, *Data Accessing Methods*]

Example **(AT&T syntax)**

```
# get the pointer to top of stack
movl %esp, %eax

# get top of stack
movl (%esp), %eax

# get the value right below top of stack
movl 4(%esp), %eax
```

- each word is 4 bytes long
- stack grows downward
- `movl` — long, 32 bits
- `%eax` — extended, 32 bits
- `4(%esp)` — base pointer addressing mode.

```
4(%esp) = 4(%esp, , ) = 4 + %esp + 0*0
```

Example **(AT&T syntax)**

```
# Full example:
# load *(ebp - 4 + (edx * 4)) into eax
movl -4(%ebp, %edx, 4), %eax
# Typical example:
# load a stack variable into eax
movl -4(%ebp), %eax
# No offset:
# copy the target of a pointer into a register
movl (%ecx), %edx
# Arithmetic:
# multiply eax by 4 and add 8
leal 8(,%eax,4), %eax
# Arithmetic:
# multiply eax by 2 and add eax (i.e. multiply by 3)
leal (%eax,%eax,2), %eax
```

- *GAS Syntax* [*GAS Syntax*].
- [*Programming from the Ground Up*, Sec. 3.5, *Addressing Modes*]
- `movl` — Data operation. load data into somewhere(register/memory).
- `leal` — Address operation. load effective address. [*Programming from the Ground Up*, Appendix. B, p259].

Example — stack setup **(AT&T syntax)**

```
# Preserve current frame pointer
pushl %ebp
# Create new frame pointer pointing to current stack top
movl %esp, %ebp
# allocate 16 bytes for locals on stack
subl $16, %esp
```

2.4.1 Assembly Language Examples

Stack Setup

Before executing a function, the program
- pushes all of the parameters for the function onto the stack. Then
- issues a *call* instruction indicating which function it wishes to start. The call instruction does two things
  1. pushes the address of the next instruction (return address) onto the stack.
  2. modifies the instruction pointer (`%eip`) to point to the start of the function.

See also: [*Programming from the Ground Up*, Sec. 4.3, *Assembly-Language Functions using the C Calling Convention*, p54].

At the time the function starts...
The stack looks like this:

```
Parameter #N
...
Parameter 2
Parameter 1
Return Address <- (%esp)
```

The function initializes the `%ebp`

```
pushl %ebp
movl %esp, %ebp
```

Now the stack looks like this:

```
Parameter #N <- N*4+4(%ebp)
...
Parameter 2 <- 12(%ebp)
Parameter 1 <- 8(%ebp)
Return Address <- 4(%ebp)
Old %ebp <- (%esp) and (%ebp)
```

each parameter can be accessed using base pointer addressing mode using the `%ebp` register

The function reserves space for locals

```
subl $8, %esp
```

Our stack now looks like this:

```
Parameter #N <- N*4+4(%ebp)
...
Parameter 2 <- 12(%ebp)
Parameter 1 <- 8(%ebp)
Return Address <- 4(%ebp)
Old %ebp <- (%ebp)
Local Variable 1 <- -4(%ebp)
Local Variable 2 <- -8(%ebp) and (%esp)
```

When a function is done executing, it does three things:

1. stores its return value in `%eax`
2. resets the stack to what it was when it was called
3. `ret` — `popl %eip` #set `eip` to *return address*

```
movl %ebp, %esp
popl %ebp
ret
```

After `ret`

```
Parameter #N
...
Parameter 2
Parameter 1 <- (%esp)
```

How about the `parameters`?

- Under many calling conventions the items popped off the stack by the epilogue include the original argument values, in which case there usually are no further stack manipulations that need to be done by the caller.
- With some calling conventions, however, it is the caller's responsibility to remove the arguments from the stack after the return.

The calling code also needs to pop off all of the parameters it pushed onto the stack in order to get the stack pointer back where it was (you can also simply add `4 × number of paramters` to `%esp` using the `addl` instruction, if you don't need the values of the parameters anymore). [*Programming from the Ground Up*, p58]

See also: [*Call stack — Wikipedia, The Free Encyclopedia*, Sec. 4.3, *Return processing*]

Generating Assembly From C Code

`simple.s` **(AT&T syntax)**

`simple.c`

```
int main()
{
  return 0;
}
```

```
$ gcc -S simple.c
```

```
        .file   "simple.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        movl    $0, %eax
        popl    %ebp
        .cfi_def_cfa 4, 4
        .cfi_restore 5
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.6.3-1) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

See also:

`$ info gas 'Pseudo Ops'`

- *What is `.cfi` and `.LFE` in assembly code produced by GCC from c++ program?* [7]
  - `.LFB` — Local Function Begin
  - `.LFE` — Local Function End
- *GAS: Explanation of `.cfi_def_cfa_offset`* [8]
- *Understanding gcc generated assembly* [9]
- *Assembler directives*[10]
- *cfi directives* [11]

Outline of an Assembly Language Program

Assembler directives (Pseudo-Ops)

Anything starting with a '`.`'

`.section .data` starts the data section

`.section .text` starts the text section

`.globl SYMBOL`

`SYMBOL` is a *symbol* marking the location of a program

`.globl` makes the symbol visible to '`ld`'

`LABEL:` a *label* defines a *symbol*'s value (address)

Generating Assembly From C Code

`simple.s` **(Oversimplified)**

```
pushl   %ebp
movl    %esp, %ebp
movl    $0, %eax
popl    %ebp
ret
```

suffixes

b byte (8 bit)

s short (16 bit integer) or single (32-bit floating point)

w word (16 bit)

l long (32 bit integer or 64-bit floating point)

q quad (64 bit)

t ten bytes (80-bit floating point)

Operation Prefixes

$ constant numbers

% register

```
pushl %ebp
 movl %esp, %ebp
 subl $8, %esp
```

- These lines save the value of EBP on the stack, then

[7]`http://stackoverflow.com/questions/3564752/what-is-cfi-and-lfe-in-assembly-code-produced-by-gcc-from-c-program`

[8]`http://stackoverflow.com/questions/7534420/gas-explanation-of-cfi-def-cfa-offset`

[9]`http://stackoverflow.com/questions/8478491/understanding-base-pointer-and-stack-pointers-in-context-with-gcc-output`

[10]`http://sourceware.org/binutils/docs/as/Pseudo-Ops.html#Pseudo-Ops`

[11]`http://blog.mozilla.com/respindola/2011/05/12/cfi-directives/`

- move the value of ESP into EBP, then
- subtract 8 from ESP.
- Note that `pushl` automatically decremented ESP by the appropriate length.
- This sequence of instructions is typical at the start of a subroutine to save space on the stack for local variables; EBP is used as the base register to reference the local variables, and a value is subtracted from ESP to reserve space on the stack (since the Intel stack grows from higher memory locations to lower ones). In this case, eight bytes have been reserved on the stack.
- See also: [*Call stack — Wikipedia, The Free Encyclopedia*, *Call Stack*]

Generating Assembly From C Code

`count.c`

```
int main()
{
  int i,j=0;

  for(i=0;i<8;i++)
    j=j+i;

  return 0;
}
```

```
$ gcc -S count.c
```

`count.s` **(AT&T syntax)**

```
        .file   "count.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        movl    -4(%ebp), %eax
        addl    %eax, -8(%ebp)
        addl    $1, -4(%ebp)
.L2:
        cmpl    $7, -4(%ebp)
        jle     .L3
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.6.3-1) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Why `subl $16, %esp`? The answer is *stack alignment*. The compiler by default keeps the stack boundary aligned to 16 bytes. We can see this from the gcc man page :

```
-mpreferred-stack-boundary=num
    Attempt to keep the stack boundary aligned to a 2 raised
    to num byte boundary. If -mpreferred-stack-boundary is not specified,
    the default is 4 (16 bytes or 128 bits).
```

Generating Assembly From C Code

`count.s` **(oversimplified)**

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        movl    -4(%ebp), %eax
        addl    %eax, -8(%ebp)
        addl    $1, -4(%ebp)
.L2:
        cmpl    $7, -4(%ebp)
        jle     .L3
        movl    $0, %eax
        leave
        ret
```

```
leave:
    movl %ebp, %esp
    popl %ebp

enter:
    pushl %ebp
    movl  %esp, %ebp
```

See also:

- *Stack Explained*[12]
- *what is “stack alignment”?* [13]
- *What does it mean to align the stack?* [14]

[12]`http://sock-raw.org/netsec/stack`
[13]`http://stackoverflow.com/questions/672461/what-is-stack-alignment`
[14]`http://stackoverflow.com/questions/4175281/what-does-it-mean-to-align-the-stack`

- [*Data structure alignment — Wikipedia, The Free Encyclopedia, Data Structure Alignment*]

2.4.2 Inline Assembly

Inline Assembly

Construct

```
asm (assembler instructions
    : output operands     /* optional */
    : input operands      /* optional */
    : clobbered registers /* optional */
    );
```

Example

```
asm ("movl %eax, %ebx");
asm ("movl %eax, %ebx" :::);
```

Use `__asm__` if the keyword `asm` conflicts with something in our program:

```
__asm__ ("movl %eax, %ebx");
__asm__ ("movl %eax, %ebx" :::);
```

Why inline? ... reduces the function-call overhead. Functions are great from the point of view of program management - they make it easy to break up your program into independent, understandable, and reuseable parts. However, function calls do involve the overhead of pushing arguments onto the stack and doing the jumps (remember locality of reference - your code may be swapped out on disk instead of in memory). For high level languages, it's often impossible for compilers to do optimizations across function-call boundaries. However, some languages support inline functions or function macros. These functions look, smell, taste, and act like real functions, except the compiler has the option to simply plug the code in exactly where it was called. This makes the program faster, but it also increases the size of the code. There are also many functions, like recursive functions, which cannot be inlined because they call themselves either directly or indirectly. [*Programming from the Ground Up*, p227, *Inline Functions*]

GCC Extended Assembly In basic inline assembly, we had only instructions. In extended assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers. *It is not mandatory to specify the registers to use, we can leave that headache to GCC and that probably fit into GCC's optimization scheme better.*

If there are a total of `n` operands (both input and output inclusive), then the first output operand is numbered `0`, continuing in increasing order, and the last input operand is numbered `n-1`. [*GCC Inline Assembly HOWTO*]
`%0` the 1st output operand
`%(n-1)` the last input operand

Example: `exit(0)`

```
{
  asm("movl $1,%%eax;"     /* SYS_exit is 1 */
      "xorl %%ebx,%%ebx;"  /* Argument is in ebx, it is 0 */
      "int  $0x80"         /* Enter kernel mode */
      );
}
```

Quick System Call Review To recap — Operating System features are accessed through system calls. These are invoked by setting up the registers in a special way and issuing the instruction `int $0x80`. Linux knows which system call we want to access by what we stored in the `%eax` register. Each system call has other requirements as to what needs to be stored in the other registers. System call number `1` is the `exit()` system call, which requires the status code to be placed in `%ebx`. [*Programming from the Ground Up*, p28]

Example

```
int foo(void)
{
  int ee = 0x4000, ce = 0x8000, reg;
  __asm__ __volatile__
    (
     "movl %1, %%eax";
     "movl %2, %%ebx";
     "call setbits"  ;
     "movl %%eax, %0"
     : "=r" (reg)        // reg [param %0] is output
     : "r" (ce), "r"(ee)  // ce [param %1], ee [param %2] are inputs
     : "%eax" , "%ebx"    // %eax and % ebx got clobbered
     )
    printf("reg=%x",reg);
}
```

- `ee,ce,reg` are local variables that will be passed as parameters to the inline assembler
- "`__volatile__`" tells the compiler not to optimize the inline assembly routine
- "`r`" means *register*; It's a constraint.
- "=" denotes an output operand, and it's *write-only*
- Clobbered registers tell GCC that the value of `%eax` and `%ebx` are to be modified inside "asm", so GCC won't use these registers to store any other value.

Extended asm If in our code we touch (ie, change the contents) some registers and return from asm without fixing those changes, something bad is going to happen. This is because GCC have no idea about the changes in the register contents and this leads us to trouble, especially when compiler makes some optimization. It will suppose that some register contains the value of some variable that we might have changed without informing GCC, and it continues like nothing happened. What we can do is either use those instructions having no side effects or fix things when we quit or wait for something to crash. This is where we want some extended functionality. Extended asm provides us with that functionality. [*GCC Inline Assembly HOWTO*, Sec. 4, *Basic Inline*]

2.5 Quirky C Language Usage

Quirky C Language Usage
`asmlinkage` and `fastcall`

`asmlinkage`
`asmlinkage int sys_fork(struct pt_regs regs)`
- tells the compiler to pass parameters on the local stack.

`fastcall`
`fastcall unsigned int do_IRQ(struct pt_regs *regs)`
- tells the compiler to pass parameters in the general-purpose registers.

Macro definition:
- *`#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))`*
- *`#define fastcall __attribute__((regparm(3)))`*

What is `asmlinkage`? This is a `#define` for some gcc magic that tells the compiler that the function should not expect to find any of its arguments in registers (a common optimization), but only on the CPU's stack. Recall our earlier assertion that `system_call` consumes its first argument, the system call number, and allows up to four more arguments that are passed along to the real system call. `system_call` achieves this feat simply by leaving its other arguments (which were passed to it in registers) on the stack. All system calls are marked with the `asmlinkage` tag, so they all look to the stack for arguments. Of course, in `sys_ni_syscall`'s case, this doesn't make any difference, because `sys_ni_syscall` doesn't take any arguments, but it's an issue for most other system calls. And, because you'll be seeing `asmlinkage` in front of many other functions, I thought you should know what it was about. [*FAQ/asmlinkage*]

It is also used to allow calling a function from assembly files.

Why `asmlinkage`? Quote from *`asmlinkage` purposes and optimizations*[15]

```
> Hi everybody,
> I'm new to this group. I'm writing to make up a doubt that I have on
> the "asmlinkage" symbol.
```

[15] `http://forum.soft32.com/linux/asmlinkage-purposes-optimizations-ftopict516190.html`

```
> I know that this keyword must be written, for example, in the
> prototype of a system call in order to say to the compiler that, when
> we call this function, it must put all the parameters in the stack,
> hence avoiding optimizations, e.g. passing parameters using registers.
>
> Now my question is: why we should avoid this kind of optimizations?
> I mean, we pass parameters to a syscall by registers. Then, the
> syscall dispatcher puts all of them into the stack and finally it
> calls the real syscall code. So why we should prevent the real syscall
> code to get its parameters from the registers?

Code can internally use any calling convention it wants. It can use an
optimized calling convention. It can use a slow calling convention.
It doesn't matter, so long as all code uses the same calling convention.
When you call into code that's not being compiled along with your current
code, you have to somehow specify that you need to use the calling
convention that this code uses rather than the default calling convention
for the compiler you're using (which may or may not be the same).

When you have a boundary between different code units, such as between
user-space code and system calls or between C code and assembly code,
you must define some calling convention. To tell the compiler to use
that calling convention, you must specify some keyword.

The details of what that calling convention is or how it may or may
not differ from the calling conventions the compiler uses for the rest
of the code are irrelevant. System calls or assembly code require some
fixed calling convention, so you have to specify it somehow.
```

See also:

- [*Calling convention — Wikipedia, The Free Encyclopedia*]
- *Is “asmlinkage”required for a c function to be called from assembly?*[16]
- *asmlinkage*[17]

Quirky C Language Usage
UL

`UL` tells the compiler to treat the value as a long value.
  - This prevents certain architectures from overflowing the bounds of their datatypes.
  - Using `UL` allows you to write architecturally independent code for large numbers or long bitmasks.

Example

```
#define GOLDEN_RATIO_PRIME 0x9e370001UL
#define ULONG_MAX (~0UL)
#define SLAB_POISON 0x00000800UL /* Poison objects */
```

`~0UL` means that complement of 0 is to be interpreted as an unsigned long explicitly casted as an unsigned int.

$$\sim \mathtt{0UL} = \mathtt{0xffffffff} = 4294967295$$

Thats the largest value that can be in a 32 bit integer [*((uint32_t)(~0UL)) ??*].

Quirky C Language Usage
static inline

inline An `inline` function results in the compiler attempting to incorporate the function’s code into all its callers.
static Functions that are visible only to other functions in the same file are known as *static functions*.

Example

```
static inline void prefetch(const void *x)
```

See also:

[16] http://stackoverflow.com/questions/10060168/is-asmlinkage-required-for-a-c-function-to-be-called-from-assembly
[17] http://hi.baidu.com/fiction_junru/blog/item/75ee131e94c397c3a78669d1.html

- *Inline Functions in C*[18]
- [*Inline function — Wikipedia, The Free Encyclopedia*]

Quirky C Language Usage
`const`

`const` — read-only
`const int *x`
- a pointer to a `const` integer
- the pointer can be changed but the integer cannot

`int const *x`
- a `const` pointer to an integer
- the integer can change but the pointer cannot

Quirky C Language Usage
`volatile`

Without `volatile`

```
static int foo;

void bar(void) {
  foo = 0;
  while (foo != 255);
}

/* optimized by compiler */
void bar_optimized(void) {
  foo = 0;
  while (true);
}
```

However, `foo` might represent a location that can be changed by other elements of the computer system at any time, such as a hardware register of a device connected to the CPU.

To prevent the compiler from optimizing code, the `volatile` keyword is used:

```
static volatile int foo;
```

See also: [*Volatile (computer programming) — Wikipedia, The Free Encyclopedia*]

2.6 Miscellaneous Quirks

Miscellaneous Quirks
`__init`

```
#define __init __attribute__ ((__section__ (".init.text")))
```

- The `__init` macro tells the compiler that the associate function or variable is used only upon initialization.
- The compiler places all code marked with `__init` into a special memory section that is freed after the initialization phase ends

Example

```
static int __init batch_entropy_init(int size, struct entropy_store *r)
```

Similarly,

`__initdata, __exit, __exitdata`

See also:
- [*The Linux Kernel Module Programming Guide*, Sec. 2.4, *Hello World (part 3): The __init and __exit Macros*]
- [*GCC Manual*, Sec. 6.36, *Variable Attributes*]

2.7 A Quick Tour of Kernel Exploration Tools

Kernel Exploration Tools

objdump Display information about object files

```
$ objdump -S simple.o
$ objdump -Dslx simple.o
```

readelf Displays information about ELF files

```
$ readelf -h a.out
```

hexdump ASCII, decimal, hexadecimal, octal dump

[18] `http://www.greenend.org.uk/rjk/tech/inline.html`

```
        $ hd a.out
```

nm List symbols from object files

```
        $ nm a.out
```

See also:

- [*Executable and Linkable Format — Wikipedia, The Free Encyclopedia*]
- *Understanding ELF using readelf and objdump*[19]

How to decompress vmlinuz? To exam the kernel image with `objdump`, you have to decompress it first.

```
$ cp /boot/vmlinuz-3.7.0-rc3-next-20121029 /tmp/
$ od -A d -t x1 vmlinuz-3.7.0-rc3-next-20121029 | grep '1f 8b 08 00'
```

The output should be something like this:

```
            0016992    f3 a5 fc 5e 8d 83 b4 91 4f 00 ff e0 1f 8b 08 00
```

```
$ dd if=vmlinuz bs=1 skip=17004 | zcat > vmlinux
```

1. How did i calculated 17004 ?

    0016992 + offset of GZ signature(`1f 8b 08 00`), i.e.

    0016992 + 12

```
$ dd if=vmlinuz-3.7.0-rc3-next-20121029 bs=1 skip=17004 | zcat > vmlinux

  5233764+0 records in
  5233764+0 records out
  5233764 bytes (5.2 MB) copied

$ file vmlinux
$ objdump -f vmlinux
```

See also: `http://comments.gmane.org/gmane.linux.kernel.kernelnewbies/42926`

2.8 Kernel Speaks: Listen to Kernel Messages

`printk()` behaves almost identically to the C library `printf()` function

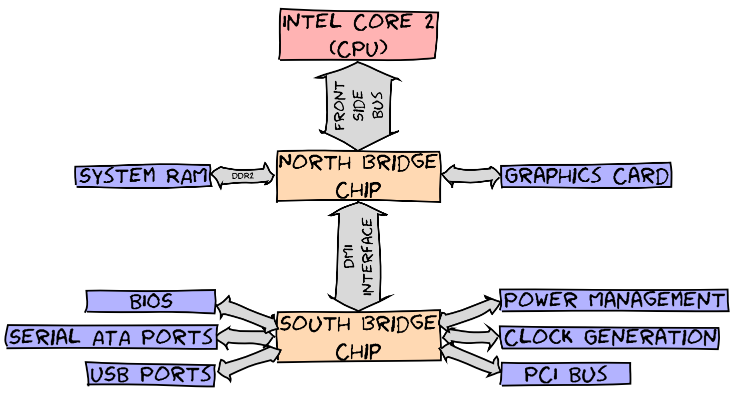
`dmesg` print or control the kernel ring buffer

`/var/log/messages` is where a majority of logged system messages reside

3 From Power Up To Bash Prompt

3.1 Motherboard Chipsets And The Memory Map

Ref: [*Motherboard Chipsets and the Memory Map*]



Facts

- The CPU doesn't know what it's connected to
    - CPU test bench? network router? toaster? brain implant?
- The CPU talks to the outside world through its pins
    - some pins to transmit the physical memory address
    - other pins to transmit the values

[19] `http://www.linuxforums.org/articles/understanding-elf-using-readelf-and-objdump_125.html`

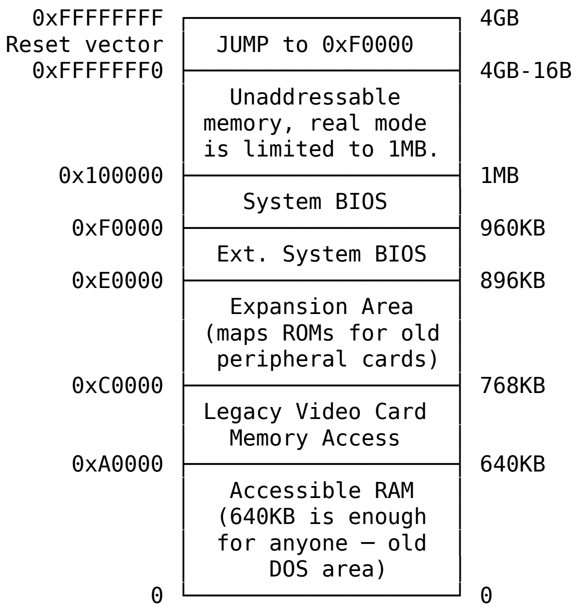
- The CPU's gateway to the world is the front-side bus

Intel Core 2 QX6600

- 33 pins to transmit the physical memory address
  - so there are 2^{33} choices of memory locations
- 64 pins to send or receive data
  - so data path is 64-bit wide, or 8-byte chunks

This allows the CPU to physically address 64GB of memory ($2^{33} \times 8B$)

See also: *Datasheet for Intel Core 2 Quad-Core Q6000 Sequence*[20]


0xFFFFFFFF
Reset vector
0xFFFFFFF0
4GB
JUMP to 0xF0000
4GB-16B
Unaddressable memory, real mode is limited to 1MB.
0x100000
1MB
System BIOS
0xF0000
960KB
Ext. System BIOS
0xE0000
896KB
Expansion Area (maps ROMs for old peripheral cards)
0xC0000
768KB
Legacy Video Card Memory Access
0xA0000
640KB
Accessible RAM (640KB is enough for anyone – old DOS area)
0
0

Some physical memory addresses are mapped away!

- only the addresses, not the spaces
- Memory holes
  - $640KB \sim 1MB$
  - /proc/iomem
- Memory-mapped I/O
  - BIOS ROM
  - video cards
  - PCI cards
  - ...

This is why 32-bit OSes have problems using 4G of RAM.

What if you don't have 4G RAM?

See also: *Memory Map (x86)*[21]

the northbridge

1. receives a physical memory request
2. decides where to route it
   - to RAM? to video card? to ...?
   - decision made via the *memory address map*
     - `/proc/iomem`
     - it is built in `setup()`

The CPU modes

real mode: CPU can only address 1MB RAM
- 20-bit address, 1-byte data unit

32-bit protected mode: can address 4GB RAM
- 32-bit address, 1-byte data unit

64-bit protected mode: can address 64GB RAM (Intel Core 2 QX6600)
- 33 address pins, 8-byte data unit

```
$ grep 'address sizes' /proc/cpuinfo
```

The *physical* stuff is determined by hard-core limits: the actual metal pins that stick out of the processor. It is *those* pins that limit the CPU to 64 gigabytes. That is completely independent of the operating system or even the mode (real-mode, 32-bit protected, 64-bit) the CPU is running in. It's a physical limit. That is the limit for which that little multiplication is done. There are 33 metal pins to transmit an address and 8 metal pins to send and receive data. So $2^{33} \times 2^3 = 2^{36} = 64GB$. The "unit"of transfer in this case is 8 bytes, that's the smallest chunk of data the CPU can address on the physical bus. In actuality, the CPU usually works in terms of cache lines, which hold 64 bytes in the Core 2s. Due to performance, the CPU reads a whole cache line at a time. So if a program reads one byte, the CPU actually reads 64 bytes and stores them in the cache. The 4-gb limit is logical, not physical. It happens because the registers and instructions in the CPU are limited to 32 bits *when it's running in 32-bit mode*,

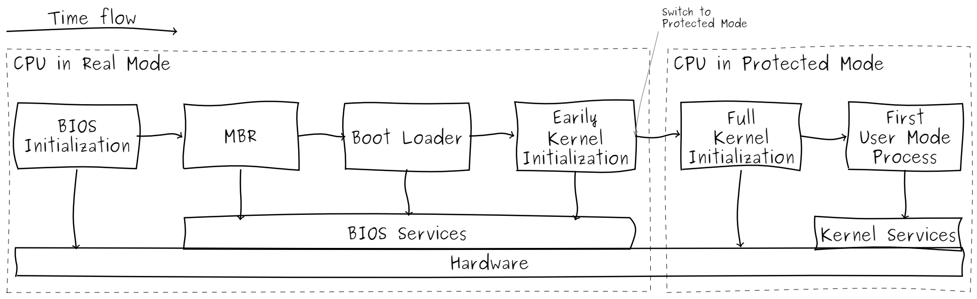
[20] `http://download.intel.com/design/processor/datashts/31559205.pdf`
[21] `http://wiki.osdev.org/Memory_Map_(x86)`

which *does* depend on the OS. Programs need to be able to address individual bytes in memory, so the "unit" of addressing is 1 byte. So *that* equation becomes $2^{32}\ addresses \times 1\ byte\ chunks = 2^{32}\ bytes$, or 4 GB total addressing. (The comments in [*Motherboard Chipsets and the Memory Map*])

3.2 How Computers Boot Up

Ref: [*How Computers Boot up*]

Bootstrapping



1. bringing at least a portion of the OS into main memory, and
2. having the processor execute it
3. the initialization of kernel data structures
4. the creation of some user processes, and
5. the transfer of control to one of them

```
$ man 7 boot
```

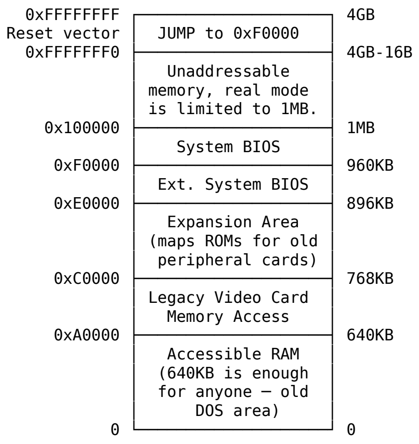
3.2.1 Motherboard power up

Motherboard power up

1. initializes motherboard firmwares (chipset, etc.)
2. gets CPU running

Real mode

CPU acts as a 1978 Intel 8086



0xFFFFFFFF
Reset vector
0xFFFFFFF0
JUMP to 0xF0000
4GB
4GB-16B
Unaddressable memory, real mode is limited to 1MB.
0x100000
1MB
System BIOS
0xF0000
960KB
Ext. System BIOS
0xE0000
896KB
Expansion Area (maps ROMs for old peripheral cards)
0xC0000
768KB
Legacy Video Card Memory Access
0xA0000
640KB
Accessible RAM (640KB is enough for anyone – old DOS area)
0
0

- any code can write to any place in memory
- only 1MB of memory can be addressed
- registers are initialized
  - EIP has `0xFFFFFFF0`, the reset vector
  - at the reset vector, there is a `jump` instruction, jumping to the *BIOS entry point* (`0xF0000`).

See also:

- [*How is the BIOS ROM mapped into address space on PC?*]
- [*System Boot Sequence*]
- [*Do normal x86 or AMD PCs run startup/BIOS code directly from ROM, or do they copy it first to RAM?*]
- [*Reset vector — Wikipedia, The Free Encyclopedia*]
- [*What Happens When A CPU Starts*]

3.2.2 BIOS

BIOS uses Real Mode addresses

- No GDT, LDT, or paging table is needed
  - the code that initializes the GDT, LDT, and paging tables must run in Real Mode

- Real mode address translation:

$$\texttt{segmentnumber} \times 2^4 + \texttt{offset}$$

e.g. to translate `<FFFF:0001>` into physical address:

$$FFFF \times 16 + 0001 = FFFF0 + 0001 = FFFF1$$

if: `offset > 0xF` (overflow)
then: `address%`2^{20} (wrap around)

– only 80286 and later x86 CPUs can address up to:

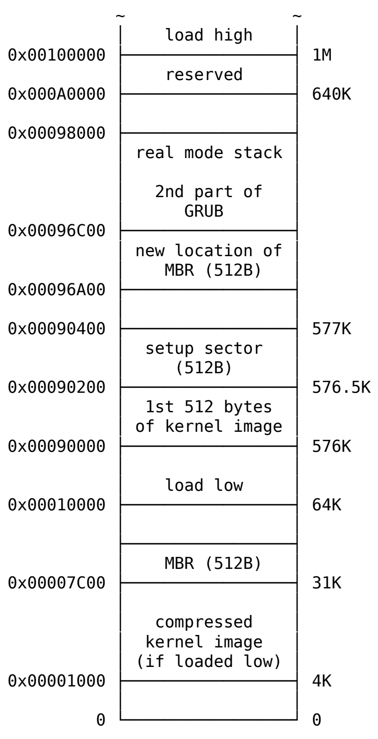
$$FFFF0 + FFFF = 10FFEF$$

See also:
- [*FreeBSD Architecture Handbook*, Sec. 1.3, *The BIOS*]
- [*Real mode — Wikipedia, The Free Encyclopedia*]
- [*Real Mode*]

CPU starts executing BIOS code

1. POST
   - an ACPI-compliant BIOS builds several tables that describe the hardware devices present in the system
2. initializes hardwares
   - at the end of this phase, a table of installed PCI devices is displayed
3. find a boot device
4. load MBR into `0x7c00`
5. Jump to `0x7c00`
6. MBR moves itself away from `0x7c00`

```
|<─────────── Master Boot Record (512 Bytes) ──────────>|
├────//────────────────────────────────//───────────────┤
│ code area │ disk-sig │ null │ partition table │ MBR-sig │
│    440    │    4     │  2   │    16x4=64      │ 0xAA55  │
└────//────────────────────────────────//───────────────┘
0          439        443    445               509      511
```



- The MBR includes a small boot loader, which is loaded into RAM starting from address `0x00007c00` by the BIOS.
  – Why `0x7c00`?[a]
- This small program
  1. moves itself to the address `0x00096a00`,
     – Why? because the boot loader may copy the boot sector of a boot partition into RAM (`0x7c00`) and execute it
  2. sets up the real mode stack (ranging from `0x00098000` to `0x000969ff`),
  3. loads the second part of the boot loader into RAM starting from address `0x00096c00`, and jumps into it.

[a] `http://www.glamenv-septzen.net/en/view/6`

3.2.3 The Boot Loader

GRUB

1. GRUB stage 1 (in MBR) loads GRUB stage 2
2. stage 2 reads GRUB configuration file, and presents boot menu

3. loads the kernel image file into memory
    - can't be done in real mode, since it's bigger than 640KB
        - BIOS supports *unreal mode*
    - 1st 512 bytes — `INITSEG, 0x00090000`
    - `setup()` — `SETUPSEG, 0x00090200`
    - load low — `SYSSEG, 0x00010000`
    - load high — `0x00100000`
4. jumps to the kernel entry point (`jmp trampoline`)
    - line 80 in `2.6.11/arch/i386/boot/setup.S`

"`jmp trampoline`" was used in 2.6.11 for calling `start_of_setup`. In newer kernels, a *2-byte jump* is used instead.

See also:

- *GRUB bootloader - Full tutorial*[22]
- Kernel source v2.6.34: *`header.S`*[23]
- *Using SHORT (Two-byte) Relative Jump Instructions*[24]
- *2-byte jump in `header.S`*[25]

More about unreal mode Unreal Mode is a "mode" where the processor runs in real mode while the segment limit does not equal 64KB (in most cases, its 4GB). Since real mode doesn't update the limit field (of the cache), this state persists across segment register loads. Entering this mode is achieved easily by entering protected mode (where the limit can be changed), load the desired limit into the descriptor cache, then switch back to real mode. [*Descriptor Cache*]

The benefits of unreal mode are quite well known: access to the 32-bit address space while simultaneously being able to call BIOS and real mode programs. [*Unreal mode*]

See also:

- A great post in OSDev forum: *Unreal mode*[26] that deserves a detailed look.
- OSDev: *Segment Registers: Real mode vs. Protected mode* covers *unreal mode, NULL selector, mode switching*[27].
- [*Segment Registers: Real mode vs. Protected mode*]
- [*Unreal mode — Wikipedia, The Free Encyclopedia*]
- [*Unreal Mode*]

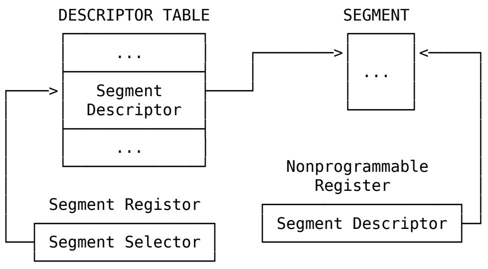


Fig. 5: Descriptor cache register

More about *bootloader*:

1. display a "Loading" message
2. load an initial portion of the kernel image from disk:
    (a) the first 512 bytes of the kernel image are put in RAM at address `0x00090000` (576K, `INITSEG`)
        - `hd -n512 /boot/vmlinuz-3.2.0-1-amd64`
        - `/usr/src/linux/arch/i386/boot/bootsect.S`
        - it was a floppy boot loader, and no longer valid since 2.6
        - nowadays to make a bootable floppy, you have to use a bootloader as you do with a hard disk

    (b) the code of the `setup()` function (see below) is put in RAM starting from address `0x00090200` ($576K + 512$, `SETUPSEG`)

[22]http://www.dedoimedo.com/computers/grub.html
[23]http://lxr.linux.no/linux+v2.6.34/arch/x86/boot/header.S#L112
[24]http://thestarman.pcministry.com/asm/2bytejumps.htm
[25]http://www.groad.net/bbs/read.php?tid-3001.html
[26]http://forum.osdev.org/viewtopic.php?f=1&t=21179&start=15
[27]http://files.osdev.org/mirrors/geezer/johnfine/segments.htm

3. load the rest of the kernel image from disk and puts the image in RAM starting from either low address 0x00010000 (64K, SYSSEG) (for small kernel images ($< 512K$) compiled with "`make zImage`") or high address 0x00100000 (1M)(for big kernel images ($> 512K$) compiled with `make bzImage`).
   ISA hole: Physical addresses ranging from 0x000a0000 (640K) to 0x000fffff ($1M - 1$) are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards
4. Jumps to the `setup()` code. (`/usr/src/linux/arch/i386/boot/setup.S`)

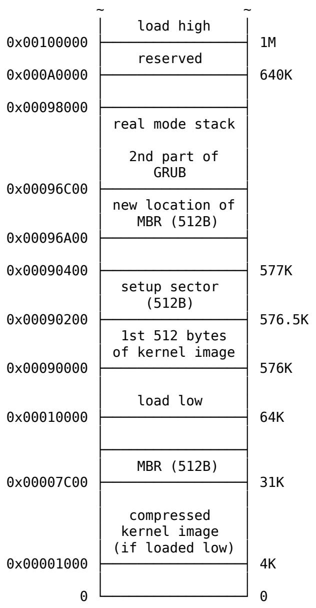
BIOS interrupt call `int $0x13` is oftenly seen in `2.6.11/arch/i386/boot/setup.S`. `INT 13H`, `INT 13h`, or `INT 19` is shorthand for BIOS interrupt call $13_{\texttt{hex}}$, the 20th interrupt vector in an x86-based computer system. The BIOS typically sets up a real mode interrupt handler at this vector that provides sector-based hard disk and floppy disk read and write services using cylinder-head-sector (CHS) addressing.

See also: [*Linux Boot Loaders Compared*, Sec. 3, *How Boot Loaders Work*]

Memory At Bootup Time

The kernel image

- `/boot/vmlinuz-x.x.x-x-x`
- has been loaded into memory by the boot loader using the BIOS disk I/O services
- The image is split into two pieces:
  - a small part containing the real-mode kernel code is loaded below the 640K barrier
  - the bulk of the kernel, which runs in protected mode, is loaded after the first megabyte of memory



See also: [*The Linux Boot Protocol, boot-time memory arrangement*].

3.3 The Kernel Boot process

3.3.1 `setup()`

The `setup()` Function

boots and loads the executable image to (0x9000 $\ll$ 4) and jumps to (0x9020 $\ll$ 4)

```
/*
 *      setup.S         Copyright (C) 1991, 1992 Linus Torvalds
 *
 * setup.s is responsible for getting the system data from the BIOS,
 * and putting them into the appropriate places in system memory.
 * both setup.s and system has been loaded by the bootblock.
 *
 * This code asks the bios for memory/disk/other parameters, and
 * puts them in a "safe" place: 0x90000-0x901FF, ie where the
 * boot-block used to be. It is then up to the protected mode
 * system to read them from there before the area is overwritten
 * for buffer-blocks.
```

- `2.6.11/arch/i386/boot/setup.S`
- Re-initialize all the hardware devices
- Sets the A20 pin (turn off *wrapping around*)
- Sets up a provisional IDT and a provisional GDT
- `PE=1, PG=0` in `cr0`
- jump to `startup_32()`

See also:

- [*From which point onwards the kernel execution starts?*]
- [*Insight into GNU/Linux boot process*]
- [*Linux Boot Process in a nutshell*]
- *Kernel attributes* are stored at the end of the boot block (1st sector). (line 90 in `bootsect.S`)

The `setup()` function details:

1. Builds system's physical memory map [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, sec. 1.1, *Memory Detection*]. See also:
   - `2.4.19/arch/i386/boot/setup.S`, line 289-389
   - `2.6.11/arch/i386/boot/setup.S`, line 302-347
   - See also: [*E820 — Wikipedia, The Free Encyclopedia*]
   - `http://www.uruk.org/orig-grub/mem64mb.html`
2. Sets the keyboard repeat delay and rate
3. Initializes the video adapter card
4. Reinitializes the disk controller and determines the hard disk parameters
5. Checks for an IBM Micro Channel bus (MCA)
6. Checks for a PS/2 pointing device (bus mouse)
7. Checks for Advanced Power Management (APM) BIOS support
8. If the BIOS supports the Enhanced Disk Drive Services (EDD), builds a table in RAM describing the hard disks available in the system
9. If the kernel image was loaded low in RAM (at physical address `0x00010000`), the function moves it to physical address `0x00001000` (was used by boot loader).

   Why? This step is necessary because to be able to store the kernel image on a floppy disk and to reduce the booting time, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM. [*Understanding The Linux Kernel*, Sec. A.3, *Middle Ages: the `setup()` Function*]
   - `BOOTSEG = 0x07C0`. This is 27K above `0x1000`. It was too small to hold the kernel image. After boot loader is done, `BOOTSEG (0x7C00)` is free. So kernel image can be stuffed here.

   Memory layout [*Insight into GNU/Linux boot process*]

   uncompressed image: ... Later, all the kernel is moved from `0x10000` (64K) to `0x1000` (4K). This move overwrites BIOS data stored in RAM, so BIOS calls can no longer be performed. We don't care because linux doesn't use BIOS to access the hardware. The first physical page is not touched because it is the so-called "zero-page", used in handling virtual memory. At this point, `setup.S` enters protected mode and jumps to `0x1000`, where the kernel lives. All the available memory can be accessed now, and the system can begin to run.
   The steps just described were once the whole story of booting when the kernel was small enough to fit in half a megabyte of memory — the address range between `0x10000` and `0x90000`. As features were added to the system, the kernel became larger than half a megabyte and could no longer be moved to `0x1000`. Thus, code at `0x1000` is no longer the Linux kernel, instead the "gunzip"part of the gzip program resides at that address.

   Compressed image (zImage): When the kernel is moved to `0x1000` (4K), `head.S` in the compressed directory is sitting at this address. It's in charge of gunzipping the kernel, this done by a function `decompress_kernel()`, defined in `compressed/misc.c`, which in turns calls `inflate()` which writes its output starting at address `0x100000` (1MB). High memory can now be accessed, because `setup.S` han take us to the protected mode now. After decompression, `head.S` jumps to the actual beginning of the kernel. The relevant code is in `../kernel/head.S`. `head.S` (i.e., the code found at `0x100000`) can complete processor initialization and call `start_ kernel()`.
   The boot steps shown above rely on the assumption that the compressed kernel can fit in half a megabyte of space. While this is true most of the time, a system stuffed with device drivers might not fit into this space. For example, kernels used in installation disks can easily outgrow the available space. To solve this problem problem bzImage kernel images were introduced.

   Big Compressed Image (bzImage): This kind of kernel image boots similarly to zImage, with a few changes.

   When the system is loaded at `0x10000` (64K) a special helper routine is called which does some special BIOS calls to move the kernel to `0x100000` (1Mb). `setup.S` doesn't move the system back to `0x1000` (4K) but, after entering protected mode, jumps instead directly to address `0x100000` (1MB) where data has been moved by the BIOS in the previous step. The decompresser found at 1MB writes the uncompressed kernel image into low memory until it is exhausted, and then into high memory after the compressed image. The two pieces are then reassembled to the address `0x100000` (1MB). Several memory moves are needed to perform the task the address `0x100000` (1MB). Several memory moves are needed to perform the task correctly.

```
code32_start: # here loaders can put a different
              # start address for 32-bit code.
#ifndef __BIG_KERNEL__
        .long 0x1000   # 0x1000 = default for zImage
#else
        .long 0x100000 # 0x100000 = default for big kernel
#endif
```

The default value of `code32` is `__BOOT_CS:0x1000` (`__BOOT_CS` = 16). (Line 855-857)

```
code32: .long 0x1000 # will be set to 0x100000 for big kernels
        .word __BOOT_CS
```

It will be changed to `__BOOT_CS:0x100000`. (Line 594-595)

```
movl %cs:code32_start, %eax
movl %eax, %cs:code32
```

`%cs:code32_start` ⇒ `%cs:code32` = `0x10:0x100000` (for load high, i.e. `bzImage`)

10. Sets the A20 pin [*A20 line — Wikipedia, The Free Encyclopedia*] located on the 8042 keyboard controller (for switching to pmode)
11. Sets up a provisional Interrupt Descriptor Table (IDT) [*Interrupt descriptor table — Wikipedia, The Free Encyclopedia*] and a provisional Global Descriptor Table (GDT) [*Global Descriptor Table — Wikipedia, The Free Encyclopedia*].
    - line 792 in `setup.S`
    - line 83-89 in `include/asm-i386/segment.h`

    The provisional GDT is created with 2 useful entries, each covering the whole 4GB address space [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.2, *Provisional GDT*]. The code that loads the GDT is:

```
xorl %eax, %eax # Compute gdt_base
movw %ds,  %ax  # (Convert %ds:gdt to a linear ptr)
shll $4,   %eax
addl $gdt, %eax
movl %eax, (gdt_48 + 2)
lgdt gdt_48     # load gdt with whatever is appropriate
```

    - `%ds = %cs = SETUPSEG = 0x9020`
    - `$gdt` — beginning address of the GDT table (somewhere offsetting in `%ds`). Its actual value will be determined at assemble time by the assembler [*Programming from the Ground Up*, p24]
    - `gdt_48`: a label in `setup.S` (Line 1006). `gdt_48+2` will be filled with the *gdt base* computed above.
    - `lgdt`: loads the value in `gdt_48` into `GDTR`
    - `gdt_48 = limit,base` ⇒ `GDTR`
        - `limit = gdt_end - gdt - 1 = 31` (16 bits)
        - `base = %ds` $\ll$ `4 + gdt` (32 bits)

```
gdt_48:
        .word gdt_end - gdt - 1 # gdt limit
        .word 0, 0              # gdt base (filled in later)

gdt:
        .fill GDT_ENTRY_BOOT_CS,8,0

        .word 0xFFFF # 4Gb - (0x100000*0x1000 = 4Gb)
        .word 0      # base address = 0
        .word 0x9A00 # code read/exec
        .word 0x00CF # granularity = 4096, 386
                     # (+5th nibble of limit)

        .word 0xFFFF # 4Gb - (0x100000*0x1000 = 4Gb)
        .word 0      # base address = 0
        .word 0x9200 # data read/write
        .word 0x00CF # granularity = 4096, 386
                     # (+5th nibble of limit)
gdt_end:
```

    - `gdt`: the provisional GDT has 4 entries.
        - The 1st and 2nd entries are initialized to 0[28] (Line 983), as required by Intel.

          `.fill GDT_ENTRY_BOOT_CS,8,0`
        - The 3rd is `__BOOT_CS`
        - The 4th is `__BOOT_DS`

      Calculate the linear base address of the kernel GDT (table) and load the GDT pointer register with its base address and limit. [*Linux 2.4.x Initialization for IA-32 HOWTO, Prepare to move to protected mode*]
        - This early kernel GDT describes kernel code as 4 GB, with base address 0, code/readable/executable, with granularity of 4 KB.
        - The kernel data segment is described as 4 GB, with base address 0, data/readable/writable, with granularity of 4 KB.
12. Resets the floating-point unit (FPU), if any.
13. Reprograms the Programmable Interrupt Controllers (PIC) to mask all interrupts, except IRQ2 which is the cascading interrupt between the two PICs.

[28] `.fill REPEAT,SIZE,VALUE`

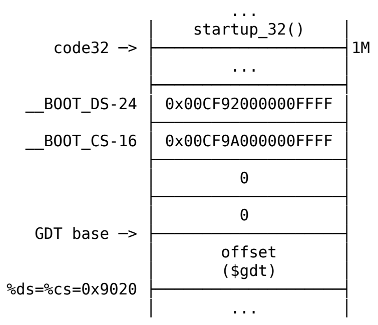


Fig. 6: Provisional GDT in RAM

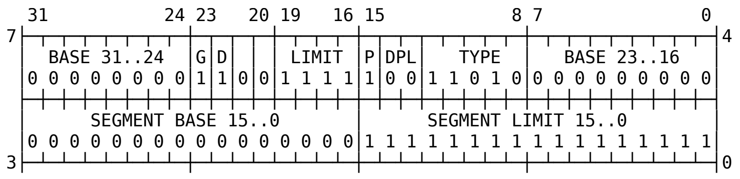


Fig. 7: Code segment descriptor value: `0x00CF9A000000FFFF`

14. Switches the CPU from Real Mode to Protected Mode by setting the `PE` bit in the `cr0` status register. The `PG` bit in the `cr0` register is cleared, so paging is still disabled. (Line 832-833)

```
movw $1, %ax # protected mode (PE) bit
lmsw %ax      # This is it!
```

   - `lmsw` — Load Machine Status Word (part of `cr0`) [*INTEL 80386 Programmer's Reference Manual*, Ch. 17]
   - in later kernel (e.g. 2.6.34) the switching code is like this (line 39-41 in pmjump.S):

```
movl %cr0, %edx
orb $X86_CR0_PE, %dl # Protected mode
movl %edx, %cr0
```

   Line 29 in `processor-flags.h`:

```
#define X86_CR0_PE 0x00000001
```

   NOTE: We are now in 32-bit protected mode. From now on, the address translation will be done by looking up the GDT table.

15. Jumps to the `startup_32()` assembly language function. (Line 854)

```
        .byte 0x66, 0xea  # prefix + jmpi-opcode
code32: .long 0x1000      # will be set to 0x100000
                          # for big kernels
        .word __BOOT_CS
```

   - `jmpi 0x100000, __BOOT_CS` (far jump)
     - jump to `0x10:0x100000` (segment number: `0x10`; offset: `0x100000`.)
   - At the end of the initial assembly code in `arch/i386/boot/setup.S` a jump to offset `0x100000` in segment `KERNEL_CS` is called. This is where the version of `startup_32()` found in `arch/i386/boot/compress-ed/hea`
     But the jump is a little tricky, as we haven't yet reloaded the `CS` register, the default size of the target offset still is 16 bit. However, using an operand prefix (`0x66`), the CPU will properly take our 48 bit far pointer [`.byte 0x66, 0xea`]. [*Insight into GNU/Linux boot process*]

3.3.2 `startup_32()`

`setup() -> startup_32()`

`startup_32()` for compressed kernel

- in `arch/i386/boot/compressed/head.S`
  - physically at
    `0x00100000` — load high, or
    `0x00001000` — load low
  - does some basic register initialization
  - `decompress_kernel()`
- the uncompressed kernel image has overwritten the compressed one starting at 1MB
- jump to the protected-mode kernel entry point at 1MB of RAM (`0x10000` $\ll$ 4)
  - `startup_32()` for real kernel

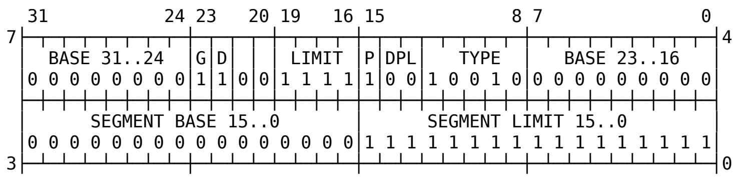


Fig. 8: Data segment descriptor value: `0x00CF92000000FFFF`

Initializing registers (Line 31 in `head.S`)

```
startup_32:
        cld
        cli
        movl $(__BOOT_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        movl %eax,%fs
        movl %eax,%gs

        lss stack_start,%esp
```

- `cld`: clear direction flag
- `cli`: clear interrupt flag
- `lss stack_start, %esp`: load `%ss` and `%esp` pair (`%ss:%esp`) in a single instruction using the value stored in `stack_start`. (Line 40 in `head.S`)
  - `lss` — read a full pointer from memory and store it in the selected segment register:register pair. [*INTEL 80386 Programmer's Reference Manual*, p332]
- `stack_start` (Line 296-303 in `misc.c`)

```
struct {
    long *a;
    short b;
} stack_start = { & user_stack [STACK_SIZE] , __BOOT_DS };
```

- `STACK_SIZE` = 4096
- `__BOOT_DS` = 24 (segment selector) ⇒ `%ss`
  - the 4th (2nd non-zero) entry in provisional GDT [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.2, *Provisional GDT*]. Each entry is 8 bytes.
- `&user_stack[4096]` ⇒ `%esp`

Clear BSS (Line 55 in `head.S`)

```
xorl %eax, %eax
movl $_edata, %edi
movl $_end, %ecx
subl %edi, %ecx
cld
rep
stosb
```

- `stosb` copies the value in `AL` into the location pointed to by `ES:DI`. `DI` is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing `AL` in the next location.
- `rep` — repeat
- For `ecx` repetitions, stores the contents of `eax` into where `edi` points to, incrementing or decrementing `edi` (depending on the direction flag) by 4 bytes each time. Normally, this is used for a memset-type operation. Usually, that instruction is simply written "`rep stosd`". Experienced assembly coders know all the details mentioned above just by seeing that. ☺
  ETA for completeness (thanks PhiS): Each iteration, `ecx` is decremented by 1, and the loop stops when it reaches zero. For `stos`, the only thing you will observe is that `ecx` is cleared at the end. [*What does `rep stos` do?*]

`startup_32()` for real kernel

`startup_32()` in `arch/i386/kernel/head.S`

- Zeroes the kernel BSS for protected mode
- sets up the final GDT
- builds provisional kernel page tables so that paging can be turned on
- enables paging (`cr3->PGDir; PG=1` in `cr0`)

- initializes a stack
- `setup_idt()` — creates the final interrupt descriptor table
- `gdtr->GDT; idtr->IDT`
- `start_kernel()`

Sets up the final GDT (line 63, 303, 448-450, 459-463, 470-EOF in `head.S`)

```
        lgdt boot_gdt_descr - __PAGE_OFFSET

        ...
        lgdt cpu_gdt_descr
        ...

boot_gdt_descr:
        .word __BOOT_DS+7                      # limit (31, end of DS)
        .long boot_gdt_table - __PAGE_OFFSET   # base location

ENTRY(boot_gdt_table)
        .fill GDT_ENTRY_BOOT_CS,8,0
        .quad 0x00cf9a000000ffff   # kernel 4GB code at 0x00000000
        .quad 0x00cf92000000ffff   # kernel 4GB data at 0x00000000

cpu_gdt_descr:
        .word GDT_ENTRIES*8-1
        .long cpu_gdt_table

        .fill NR_CPUS-1,8,0       # space for the other GDT descriptors

ENTRY(cpu_gdt_table)
        ...
        # Entry 12-15
        .quad 0x00cf9a000000ffff   # 0x60 kernel 4GB code at 0x00000000
        .quad 0x00cf92000000ffff   # 0x68 kernel 4GB data at 0x00000000
        .quad 0x00cffa000000ffff   # 0x73 user 4GB code at 0x00000000
        .quad 0x00cff2000000ffff   # 0x7b user 4GB data at 0x00000000
        ...
```

- `.fill REPEAT, SIZE, VALUE`
  - `.fill 2,8,0` — fills the 1st and 2nd entry with `0`
- `.quad` 8-byte datas (the 3rd and 4th entry)
- The addresses of `boot_gdt_table` and `cpu_gdt_table` will be assigned by assembler at compile time.

Zeroes the kernel BSS (line 74-79 in `arch/i386/kernel/head.S`)

```
xorl %eax,%eax
movl $__bss_start - __PAGE_OFFSET,%edi
movl $__bss_stop - __PAGE_OFFSET,%ecx
subl %edi,%ecx
shrl $2,%ecx
rep ; stosl
```

- `subl %edi, %ecx` — get BSS size, put into `%ecx`
- `shrl $2, %ecx` — $\frac{\%ecx}{4}$, get number of 4-byte chunks (repeat times)
- `stosl/stosd` — Store `EAX` in dword `ES:EDI`, update `EDI`

Initialize page tables [*Understanding The Linux Kernel*, Sec. 2.5.5, *Kernel Page Tables*]

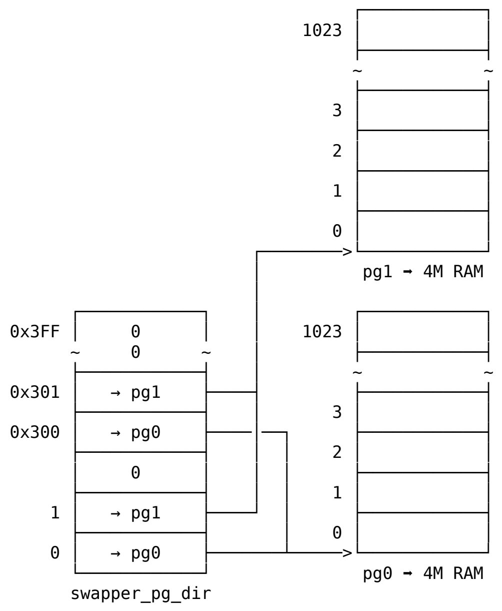
In the first phase, the kernel creates a limited address space including the kernel's code and data segments, the initial Page Tables, and 128 KB for some dynamic data structures. This minimal address space is just large enough to install the kernel in RAM and to initialize its core data structures.

The provisional Page Global Directory is contained in the `swapper_pg_dir` variable. `swapper_pg_dir` is at the beginning of BSS (uninitialized data area) because BSS is no longer used after system start up.

The provisional Page Tables are stored starting from `pg0`, right after the end of the kernel's uninitialized data segments (`_end`).

For the sake of simplicity, let's assume that the kernel's segments, the provisional Page Tables, and the 128 KB memory area fit in the first 8 MB of RAM. In order to map 8 MB of RAM, two Page Tables are required.

The objective of this first phase of paging is to allow these 8 MB of RAM to be easily addressed both in real mode and protected mode.



Therefore, the kernel must create a mapping from both the linear addresses `0x00000000` through `0x007fffff` (8M) and the linear addresses `0xc0000000` through `0xc07fffff` (8M) into the physical addresses `0x00000000` through `0x007fffff`. In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from `0xc0000000`.

Why? From [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.3.2, *Provisional Kernel Page Tables*]:

- All pointers in the compiled kernel refer to addresses $> PAGE\_OFFSET$. That is, the kernel is linked under the assumption that its base address will be `start_text` (I think; I don't have the code on hand at the moment), which is defined to be $PAGE\_OFFSET + (some\ small\ constant,\ call\ it\ C)$.
- All the kernel bootstrap code (mostly real mode code) is linked assuming that its base address is $0 + C$.

`head.S` is part of the bootstrap code. It's running in protected mode with paging turned off, so all addresses are physical. In particular, the instruction pointer is fetching instructions based on physical address. The instruction that turns on paging (`movl %eax, %cr0`) is located, say, at some physical address `A`.

As soon as we set the paging bit in `cr0`, paging is enabled, and starting at the very next instruction, all addressing, including instruction fetches, pass through the address translation mechanism (page tables). IOW, all address are henceforth virtual. That means that

1. We must have valid page tables, and
2. Those tables must properly map the instruction pointer to the next instruction to be executed.

That next instruction is physically located at address `A+4` (the address immediately after the "`movl %eax, %cr0`" instruction), but from the point of view of all the kernel code — which has been linked at `PAGE_OFFSET` — that instruction is located at virtual address `PAGE_OFFSET+(A+4)`. Turning on paging, however, does not magically change the value of EIP [29]. The CPU fetches the next instruction from ***virtual*** address `A+4`; that instruction is the beginning of a short sequence that effectively relocates the instruction pointer to point to the code at `PAGE_OFFSET + A + (something)`.

But since the CPU is, for those few instructions, fetching instructions based on physical addresses ***but having those instructions pass through address translation***, we must ensure that both the physical addresses and the virtual addresses are :

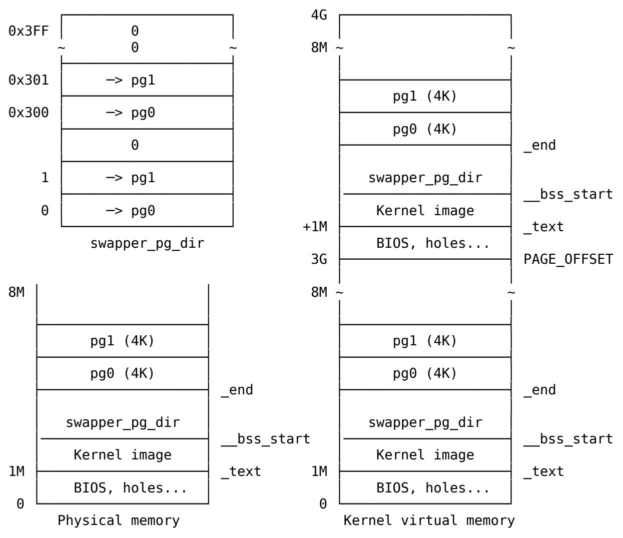
1. Valid virtual addresses, and
2. Point to the same code.

That means that at the very least, the initial page tables must

1. map virtual address `PAGE_OFFSET+(A+4)` to physical address `(A+4)`, and must
2. map virtual address `A+4` to physical address `A+4`.

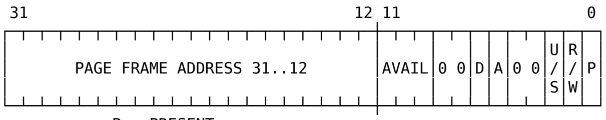
This dual mapping for the first 8MB of physical RAM is exactly what the initial page tables accomplish. The 8MB initally mapped is more or less arbitrary. It's certain that no bootable kernel will be greater than 8MB in size. The identity mapping is discarded when the MM system gets initialized.

[29] The value of EIP is still physically `A+4`, not `PAGE_OFFSET+(A+4)` yet. But since paging is just enabled, CPU could pass `A+4` through address translation.



Line 91-111 in `head.S`

```
        page_pde_offset = (__PAGE_OFFSET >> 20);

            movl $(pg0 - __PAGE_OFFSET), %edi
            movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
            movl $0x007, %eax                   # 0x007 = PRESENT+RW+USER
        10:
            leal 0x007(%edi),%ecx               # Create PDE entry
            movl %ecx,(%edx)                    # Store identity PDE entry
            movl %ecx,page_pde_offset(%edx)     # Store kernel PDE entry
            addl $4,%edx
            movl $1024, %ecx
        11:
            stosl
            addl $0x1000,%eax
            loop 11b
            # End condition: we must map up to and including INIT_MAP_BEYOND_END
            # bytes beyond the end of our own page tables; the +0x007 is the attribute bits
            leal (INIT_MAP_BEYOND_END+0x007)(%edi),%ebp
            cmpl %ebp,%eax
            jb 10b
            movl %edi,(init_pg_tables_end - __PAGE_OFFSET)
```



```
      P - PRESENT
    R/W - READ/WRITE
    U/S - USER/SUPERVISOR
      A - ACCESSED
      D - DIRTY
  AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

  NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

- `page_pde_offset = (__PAGE_OFFSET >> 20);` *`/* 0xC00, the 3K point. */`*
  The `PGDir` is one page (4K) in size. It's divided into two parts:
  1. first 3K (768 entries) for user mode
  2. last 1K (256 entries) for kernel mode
- `$(pg0 - __PAGE_OFFSET)` yields the physical address of `pg0` since here it is a linear address. Same case for `$(swapper_pg_dir - __PAGE_OFFSET)`.
  - `swapper_pg_dir` starts at the beginning of BSS
  - `pg0` starts at `_end`
- Registers:
  `%edi` address of each page table entry, i.e. `pg0[0]..pg0[1023]`, `pg1[0]..pg1[1023]`.
  `%edx` address of `swapper_pg_dir[0]`, and then to `swapper_pg_dir[1]`.
  `%ecx` has two uses
  1. contents of `swapper_pg_dir[0]`, `swapper_pg_dir[1]`, `swapper_pg_dir[768]`, `swapper_pg_dir[769]`.
  2. loop counter (1024 -> 0)

  `%eax` `7, 4k+7, 8k+7 ... 8M-4k+7` for 2k page table entries in `pg0` and `pg1` respectively.
  `%ebp` `= 128k + 7 + &pg0[1023]` in the first round of loop. Its value cannot be determined at coding time, because the address of `pg0` is not known until compile/link time.
- `stosl`: stores the contents of EAX at the address pointed by `EDI`, and increments `EDI`. Equivalent to:

```
movl %eax, (%edi)
addl $4, %edi
```

- `cmpl, jb`: if `%eax` < `%ebp`, jump to 10;
  - `jb`: jump on below/less than, unsigned [*X86 assembly language — Wikipedia, The Free Encyclopedia*, Sec. 8, *Program flow*]
  - At the end of the 1st round of loop, the value of `%eax` is `4M-4k+7`, while the value of `%ebp` depends on the address of `pg0`.
    If the kernel image is small enough (e.g. a `zImage`), `pg0` could be low enough to let the 128KB covered without a `pg1`.
- `INIT_MAP_BEYOND_END`: 128KB, used as a bitmap covering all pages. For 1M pages (4GB RAM), we need 1M bits (128K bytes)[30]

Equivalent pseudo C code[31]:

```
/*
 * Provisional PGDir and page tables setup
 *
 * for mapping two linear address ranges to the same physical address range
 *
 *   + Linear address ranges:
 *             -   User mode: $i\times{}4M\sim{}(i+1)\times{}4M-1$
 *             - Kernel mode: $3G+i\times{}4M\sim{}3G+(i+1)\times{}4M-1$
 *   + Physical address range: $i\times{}4M\sim{}(i+1)\times{}4M-1$
 */
typedef unsigned int PTE;
PTE *pg = pg0;      /* physical address of pg0 */
PTE pte = 0x007;    /* 0x007 = PRESENT+RW+USER */
for(i=0;;i++){
  swapper_pg_dir[i] = pg + 0x007;          /* store identity PDE entry */
  swapper_pg_dir[i+page_pde_offset] = pg + 0x007; /* kernel PDE entry */
  for(j=0;j<1024;j++){                /* populating one page table */
    pg[i*1024 + j] = pte;             /* fill up one page table entry */
    pte += 0x1000;                    /* next 4k */
  }
  if(pte >= ((char*)pg + i*1024 + j)*4 + 0x007 + INIT_MAP_BEYOND_END)
    {
      init_pg_tables_end = pg + i*0x1000 + j;
      break;
    }
  }
```

Enable paging [*Linux i386 Boot Code HOWTO*, Sec. 6.1] (Line 186-194 in `head.S`)

```
# Enable paging
movl $swapper_pg_dir - __PAGE_OFFSET, %eax
movl %eax, %cr3  # set the page table pointer..
movl %cr0, %eax
 orl $0x80000000, %eax
movl %eax, %cr0  # ..and set paging (PG) bit
```

3.3.3 `start_kernel()`

`start_kernel()` — a long list of calls to initialize various kernel subsystems and data structures

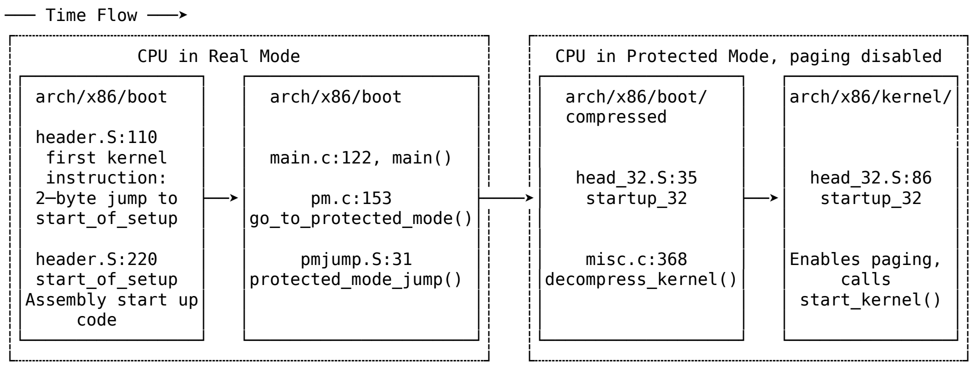
- `sched_init()` — scheduler
- `build_all_zonelists()` — memory zones
- `page_alloc_init(), mem_init()` — buddy system
- `trap_init(), init_IRQ()` — IDT
- `time_init()` — time keeping
- `kmem_cache_init()` — slab allocator
- `calibrate_delay()` — CPU clock
- `kernel_thread()` — The kernel thread for process 1
- login prompt

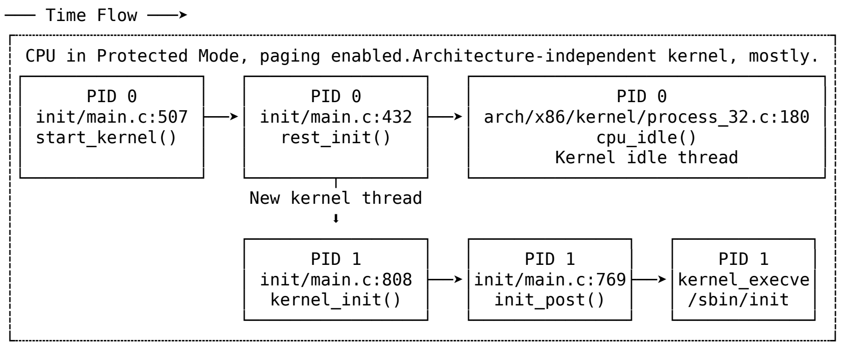
The Kernel Boot Process

$$\begin{tikzcd}
\boxed{\texttt{BIOS}} \arrow[r] & \boxed{\texttt{Bootloader}} \arrow[r] & \boxed{\texttt{setup()}} \arrow[r] & \boxed{\texttt{startup\_32()}} \arrow[r] & \boxed{\texttt{startup\_32()}} \arrow[r] & \boxed{\texttt{start\_kernel()}} \\
 & & \texttt{PE=1;PG=0} & & \texttt{PG=1} &
\end{tikzcd}$$

[30]dead link: `http://kerneldiy.com/blog/?p=201`
[31]`http://www.eefocus.com/article/09-04/71517s.html`

The Kernel Boot Process


Time Flow
CPU in Real Mode
arch/x86/boot
header.S:110 first kernel instruction: 2–byte jump to start_of_setup
header.S:220 start_of_setup Assembly start up code
arch/x86/boot
main.c:122, main()
pm.c:153 go_to_protected_mode()
pmjump.S:31 protected_mode_jump()
CPU in Protected Mode, paging disabled
arch/x86/boot/compressed
head_32.S:35 startup_32
misc.c:368 decompress_kernel()
arch/x86/kernel/
head_32.S:86 startup_32
Enables paging, calls start_kernel()

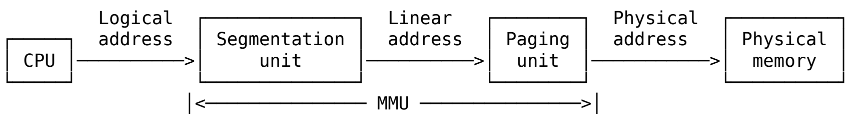

Time Flow
CPU in Protected Mode, paging enabled.Architecture-independent kernel, mostly.
PID 0 init/main.c:507 start_kernel()
PID 0 init/main.c:432 rest_init()
PID 0 arch/x86/kernel/process_32.c:180 cpu_idle() Kernel idle thread
New kernel thread
PID 1 init/main.c:808 kernel_init()
PID 1 init/main.c:769 init_post()
PID 1 kernel_execve /sbin/init

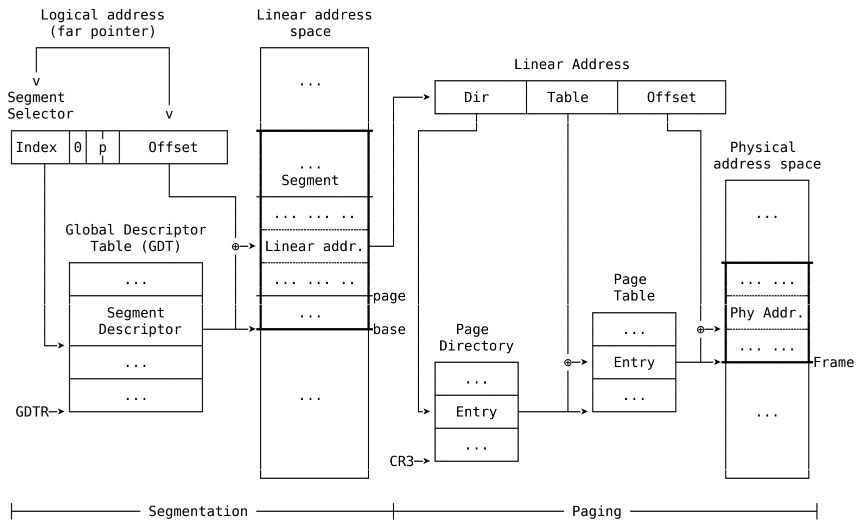
- This picture is not fully compatible with 2.6.11.

4 Memory Addressing

4.1 Memory Addresses

Three Kinds Of Addresses





All CPUs Share The Same Memory

Memory Arbiter

if:	the chip is free
then:	grants access to a CPU
if:	the chip is busy servicing a request by another processor

then: delay it

Even uniprocessor systems use memory arbiters because of *DMA*.

4.2 Segmentation in Hardware

Real Mode Address Translation

- Backward compatibility of the processors
- BIOS uses real mode addressing
- Use 2 16-bit registers to get a 20-bit address

Logical address format

`<segment:offset>`

Real mode address translation

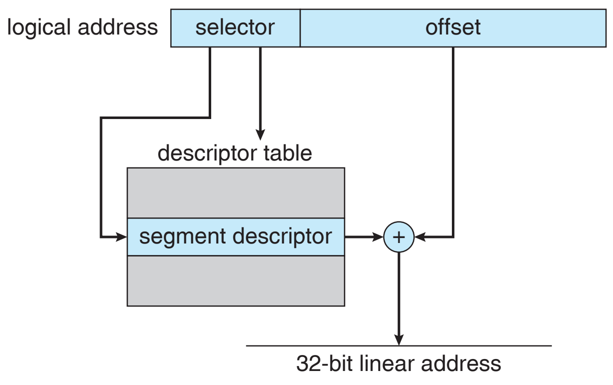
$$\texttt{segment number} \times 2^4 + \texttt{offset}$$

e.g. to translate <`FFFF:0001`> into linear address:

$$\texttt{FFFF} \times 16 + 0001 = \texttt{FFFF0} + 0001 = \texttt{FFFF1}$$

See also: [汇编语言, Sec. 2.4, *16-bit CPU*]

Protected Mode Address Translation

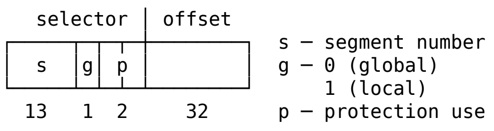


Segment Selectors

A logical address consists of two parts:

segment selector	:	offset
16 bits		32 bits

Segment selector is an index into GDT/LDT



Segmentation Registers

Segment registers hold segment selectors

cs code segment register

CPL 2-bit, specifies the Current Privilege Level of the CPU

00 - Kernel mode

11 - User mode

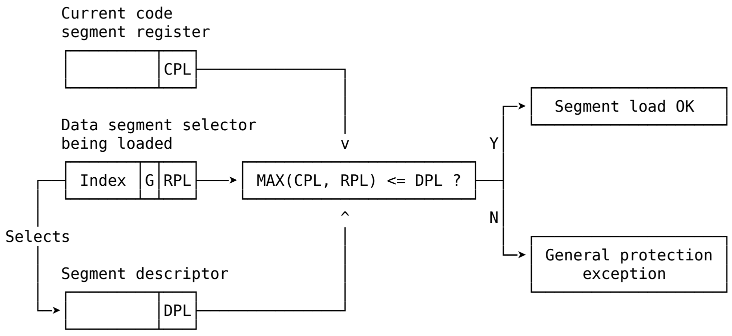
ss stack segment register

ds data segment register

es/fs/gs general purpose registers, may refer to arbitrary data segments

See also:

- [*INTEL 80386 Programmer's Reference Manual*, Sec. 6.3, *Segment-Level Protection*]
- [*CPU Rings, Privilege, and Protection*]


Current code segment register
CPL
Data segment selector being loaded
Index
G
RPL
MAX(CPL, RPL) <= DPL ?
Y
N
Segment load OK
General protection exception
Selects
Segment descriptor
DPL

Segment Descriptors

All the segments are organized in 2 tables:

GDT *Global Descriptor Table*

- shared by all processes
- GDTR stores address and size of the GDT

LDT *Local Descriptor Table*

- one process each
- LDTR stores address and size of the LDT

Segment descriptors are entries in either GDT or LDT, 8-byte long

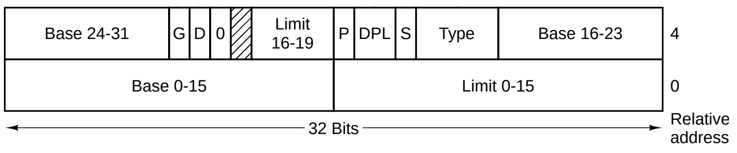
Analogy

Process	$\Longleftrightarrow$	Process Descriptor(PCB)
File	$\Longleftrightarrow$	Inode
Segment	$\Longleftrightarrow$	Segment Descriptor

See also:

- Memory Translation And Segmentation [*Memory Translation and Segmentation*]
- `http://www.osdever.net/bkerndev/Docs/gdt.htm`
- Sec 4 of *JamesM's kernel development tutorials*[32], The GDT and IDT

Example: A LDT entry for code segment



Base: Where the segment starts

Limit: 20 bit, $\Rightarrow 2^{20}$ in size

G: Granularity flag
- 0 - segment size in bytes
- 1 - in 4096 bytes

S: System flag
- 0 - system segment, e.g. LDT
- 1 - normal code/data segment

D/B:
- 0 - 16-bit offset
- 1 - 32-bit offset

Type: segment type (cs/ds/tss)
- TSS: Task status, i.e. it's executing or not

DPL: Descriptor Privilege Level. 0/3

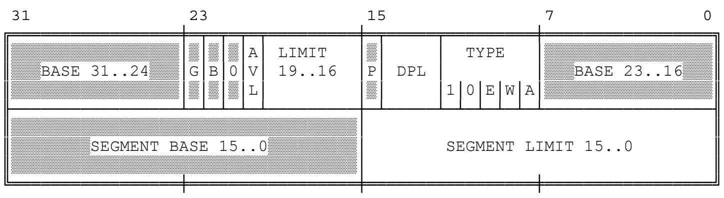
P: Segment-Present flag
- 0 - not in memory
- 1 - in memory

AVL: ignored by Linux

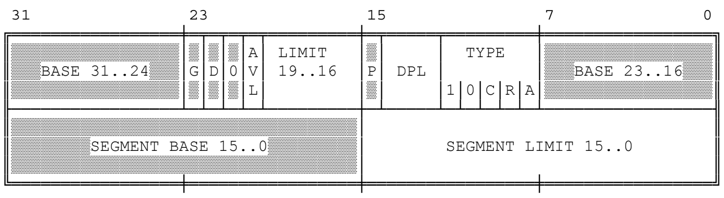
See: [*INTEL 80386 Programmer's Reference Manual*, Sec. 6.3, *Segment-Level Protection*, p108]

[32]`http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html`

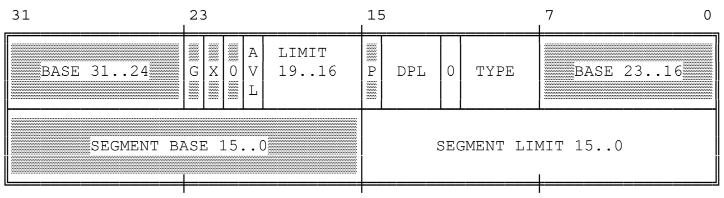
DATA SEGMENT DESCRIPTOR



EXECUTABLE SEGMENT DESCRIPTOR



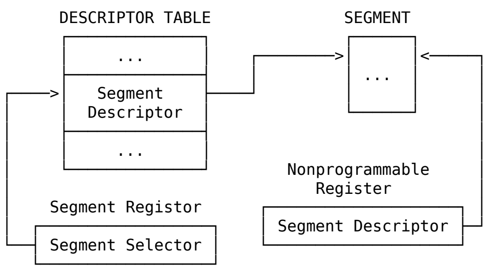
SYSTEM SEGMENT DESCRIPTOR



A	- ACCESSED	E	- EXPAND-DOWN
AVL	- AVAILABLE FOR PROGRAMMERS USE	G	- GRANULARITY
B	- BIG	P	- SEGMENT PRESENT
C	- CONFORMING	R	- READABLE
D	- DEFAULT	W	- WRITABLE
DPL	- DESCRIPTOR PRIVILEGE LEVEL		

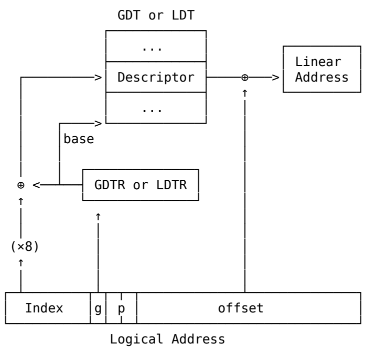
Fast Access to Segment Descriptors

a non-programmable cache register for each segment register



Translating a logical address

1. $Index \times 8 + table\ base$
2. $Descriptor\ base + offset$



4.3 Segmentation in Linux

Linux prefers paging to segmentation

Because

- Segmentation and paging are somewhat redundant
- Memory management is simpler when all processes share the same set of linear addresses

- Maximum portability. RISC architectures in particular have limited support for segmentation

The Linux 2.6 uses segmentation only when required by the 80x86 architecture.

The Linux GDT Layout

Each GDT includes 18 segment descriptors and 14 null, unused, or reserved entries

`include/asm-i386/segment.h`

0	null	8	TLS segment #3	16	TSS	24	APM BIOS support
1	reserved	9	reserved	17	LDT	25	APM BIOS support
2	reserved	10	reserved	18	PNPBIOS support	26	ESPFIX small SS
3	reserved	11	reserved	19	PNPBIOS support	27	per-cpu
4	unused	12	kernel code segment	20	PNPBIOS support	28	stack_canary-20
5	unused	13	kernel data segment	21	PNPBIOS support	29	unused
6	TLS segment #1	14	default user CS	22	PNPBIOS support	30	unused
7	TLS segment #2	15	default user DS	23	APM BIOS support	31	TSS for double fault handler

Although Linux doesn't use hardware context switches, it is nonetheless forced to set up a TSS for each distinct CPU in the system [*Understanding The Linux Kernel*, Sec. 3.3.2, *Task State Segment*]. This is done for two main reasons:

- When an 80x86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS (see the sections "*Hardware Handling of Interrupts and Exceptions*" in Chapter 4 and "*Issuing a System Call via the sysenter Instruction*" in Chapter 10) [*Task state segment — Wikipedia, The Free Encyclopedia*, Sec. 5, *Inner-level stack pointers*].
- When a User Mode process attempts to access an I/O port by means of an `in` or `out` instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port. [*Task state segment — Wikipedia, The Free Encyclopedia*, Sec. 4, *I/O port permissions*]

See also:

- [*Global Descriptor Table — Wikipedia, The Free Encyclopedia*]
- [*Understanding The Linux Kernel*, Sec. 2.3.2, *The Linux LDTs*]

The Four Main Linux Segments

Every process in Linux has these 4 segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

All linear addresses start at 0, end at 4G-1

- All processes share the same set of linear addresses
- Logical addresses coincide with linear addresses

Segment Selectors

`include/asm-i386/segment.h`

```
#define GDT_ENTRY_DEFAULT_USER_CS        14
#define __USER_CS (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)

#define GDT_ENTRY_DEFAULT_USER_DS        15
#define __USER_DS (GDT_ENTRY_DEFAULT_USER_DS * 8 + 3)

#define GDT_ENTRY_KERNEL_BASE    12

#define GDT_ENTRY_KERNEL_CS              (GDT_ENTRY_KERNEL_BASE + 0)
#define __KERNEL_CS (GDT_ENTRY_KERNEL_CS * 8)

#define GDT_ENTRY_KERNEL_DS              (GDT_ENTRY_KERNEL_BASE + 1)
#define __KERNEL_DS (GDT_ENTRY_KERNEL_DS * 8)
```

$Selector = Index \ll 3 + G + RPL$

`__USER_CS`	$14 \ll 3 + 3 = 115$	0000 0000 0111 0011
`__USER_DS`	$15 \ll 3 + 3 = 123$	0000 0000 0111 1011
`__KERNEL_CS`	$12 \ll 3 + 0 = 96$	0000 0000 0110 0000
`__KERNEL_DS`	$13 \ll 3 + 0 = 104$	0000 0000 0110 1000

Values in segment registers:

Index	G:RPL	
0 0 0 0 0 0 0 0 0 0 1 1 1 0	0 1 1	`__USER_CS   (14 « 3 + 3)`
0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 1 1	`__USER_DS   (15 « 3 + 3)`
0 0 0 0 0 0 0 0 0 0 1 1 0 0	0 0 0	`__KERNEL_CS (12 « 3 + 0)`
0 0 0 0 0 0 0 0 0 0 1 1 0 1	0 0 0	`__KERNEL_DS (13 « 3 + 0)`

The corresponding Segment Selectors are defined by the macros `__USER_CS`, `__USER_DS`, `__KERNEL_CS`, and `__KERNEL_DS`, respectively. To address the kernel code segment, for instance, the kernel just loads the value yielded by the `__KERNEL_CS` macro into the `cs` segmentation register [*Understanding The Linux Kernel*, Sec. 2.3, *Segmentation in Linux*].

Notice that the linear addresses associated with such segments all start at `0` and reach the addressing limit of $2^{32} - 1$. This means that all processes, either in User Mode or in Kernel Mode, may use the same logical addresses.

Another important consequence of having all segments start at `0x00000000` is that in Linux, logical addresses coincide with linear addresses; that is, the value of the Offset field of a logical address always coincides with the value of the corresponding linear address.

As stated earlier, the Current Privilege Level of the CPU indicates whether the processor is in User or Kernel Mode and is specified by the RPL field of the Segment Selector stored in the `cs` register. *Whenever the CPL is changed, some segmentation registers must be correspondingly updated.* For instance, when the CPL is equal to 3 (User Mode), the `ds` register must contain the Segment Selector of the user data segment, but when the CPL is equal to 0, the `ds` register must contain the Segment Selector of the kernel data segment.

A similar situation occurs for the `ss` register. It must refer to a User Mode stack inside the user data segment when the CPL is 3, and it must refer to a Kernel Mode stack inside the kernel data segment when the CPL is 0. *When switching from User Mode to Kernel Mode, Linux always makes sure that the* `ss` *register contains the Segment Selector of the kernel data segment.*

When saving a pointer to an instruction or to a data structure, the kernel does not need to store the Segment Selector component of the logical address, because the `ss` register contains the current Segment Selector. As an example, when the kernel invokes a function, it executes a `call` assembly language instruction specifying just the Offset component of its logical address; the Segment Selector is implicitly selected as the one referred to by the `cs` register. Because there is just one segment of type ”executable in Kernel Mode,” namely the code segment identified by `__KERNEL_CS`, it is sufficient to load `__KERNEL_CS` into `cs` whenever the CPU switches to Kernel Mode. The same argument goes for pointers to kernel data structures (implicitly using the `ds` register), as well as for pointers to user data structures (the kernel explicitly uses the `es` register).

Example:
To address the kernel code segment, the kernel just loads the value yielded by the `__KERNEL_CS` macro into the `cs` segmentation register.

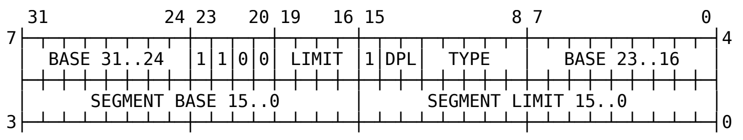
Note that

1. `base = 0`
2. `limit = 0xfffff`

This means that

- all processes, either in User Mode or in Kernel Mode, may use the same logical addresses
- logical addresses (Offset fields) coincide with linear addresses

The Linux GDT
8 byte segment descriptor



`arch/i386/kernel/head.S`

```
ENTRY(cpu_gdt_table)
 .quad 0x00cf9a000000ffff /* 0x60 kernel 4GB code at 0x00000000 */
 .quad 0x00cf92000000ffff /* 0x68 kernel 4GB data at 0x00000000 */
 .quad 0x00cffa000000ffff /* 0x73 user 4GB code at 0x00000000 */
 .quad 0x00cff2000000ffff /* 0x7b user 4GB data at 0x00000000 */
```

- 0x60 $= 12 \ll 3 + 0$
- 0x68 $= 13 \ll 3 + 0$
- 0x73 $= 14 \ll 3 + 3$
- 0x7b $= 15 \ll 3 + 3$

From the comments of Memory Translation And Segmentation [*Memory Translation and Segmentation*]:

Q: I went to where the `gdt_page` is instantiated (Line 24, `common.c`, 2.6.25)
It has the following code:
`[GDT_ENTRY_DEFAULT_USER_CS] =    0x0000ffff, 0x00cffa00`
Do you know what that means?

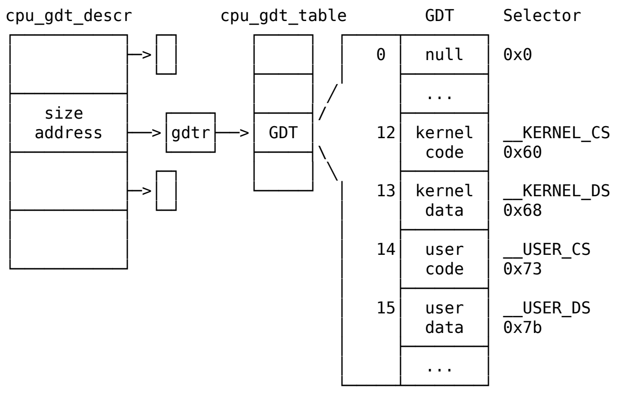
A: This line is building the 8-byte segment descriptor for the user code segment. To really follow it, there are 3 things you must bear in mind:

1. The x86 is little endian, meaning that for multi-byte data types (say, 32-bit or 64-bit integers), the significance of bytes grows with memory address. If you declare a 32-bit integer as `0xdeadbeef`, then it would be laid out in memory like this (in hex, assuming memory addresses are growing to the right):
```
ef be ad de
lower => higher
```
2. In array declarations, or in this case a struct declaration, earlier elements go into lower memory addresses.
3. The convention for Intel diagrams is to draw things with HIGHER memory addresses on the LEFT and on TOP. This is a bit counter intuitive, but I followed it to be consistent with Intel docs.

When you put this all together, the declaration above will translate into the following bytes in memory, using Intel's 'reversed' notation:

```
(higher)
00 cf fa 00
00 00 ff ff
(lower)
```

If you compare these values against the segment descriptor diagram above, you'll see that: the 'base' fields are all zero, so the segment starts at `0x00000000`, the limit is `0xfffff` so the limit covers 4GB, byte 47-40 is `11111010`, so the DPL is 3 for ring 3.
If you look into the Intel docs, they describe the fields I left grayed out. Hope this helps!



cpu_gdt_table: is an array of all GDTs
cpu_gdt_descr: store the addresses and sizes of the GDTs

4.4 Paging in Hardware

Paging in Hardware
Starting with the 80386, all 80x86 processors support paging

A page is
- a set of linear addresses
- a block of data

A page frame is

- a constituent of main memory
- a storage area

A page table

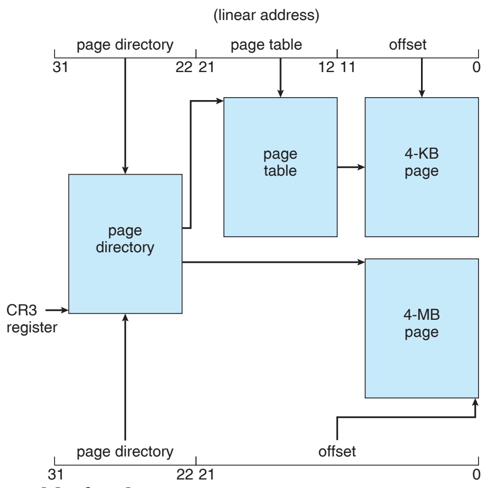
- is a data structure
- maps linear to physical addresses
- stored in main memory

Pentium Paging

Linear Address ⇒ Physical Address

Two page size in Pentium:
4K: 2-level paging
4M: 1-level paging

```
      page number      | page offset
+----------+----------+------------+
|    p1    |    p2    |     d      |
+----------+----------+------------+
    10         10           12
     |          |
     |          `-> pointing to 1k frames
     `--> pointing to 1k page tables
```

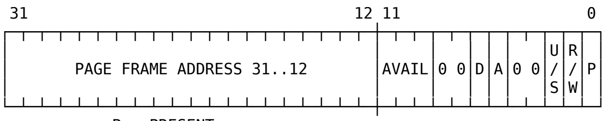


- The CR3 register points to the top level page table for the current process.

Same structure for Page Dirs and Page Tables

- 4 bytes (32 bits) long
- Page size is usually 4k (2^{12} bytes). OS dependent
  `$ getconf PAGESIZE`
- Could have $2^{32-12} = 2^{20} = 1M$ pages
  Could addressing $1M \times 4KB = 4GB$ memory

Intel i386 page table entry



```
    P — PRESENT
  R/W — READ/WRITE
  U/S — USER/SUPERVISOR
    A — ACCESSED
    D — DIRTY
AVAIL — AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

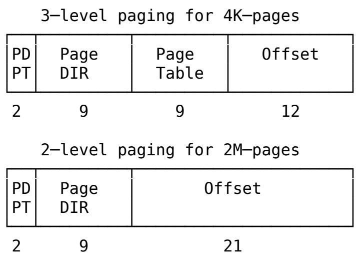
Physical Address Extension (PAE)

32-bit linear⇒ 36-bit physical

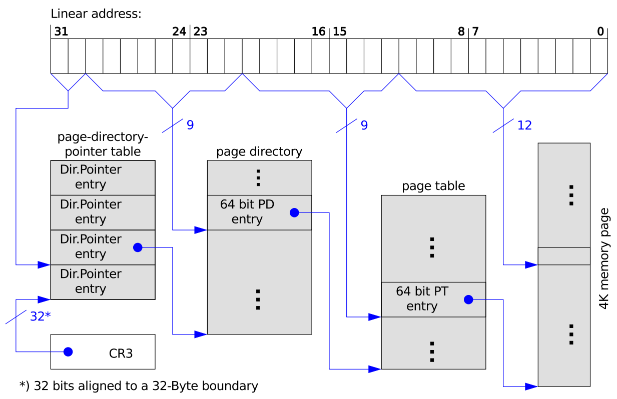
Need a new paging mechanism

	Linear Address	Physical Address	Max RAM	Page Size	PTE Size	Paging Level
No PAE	32 bits	32 bits	$2^{32} = 4GB$	4K,4M	32 bits	1,2
PAE	32 bits	36 bits	$2^{36} = 64GB$	4K,2M	64 bits	2,3

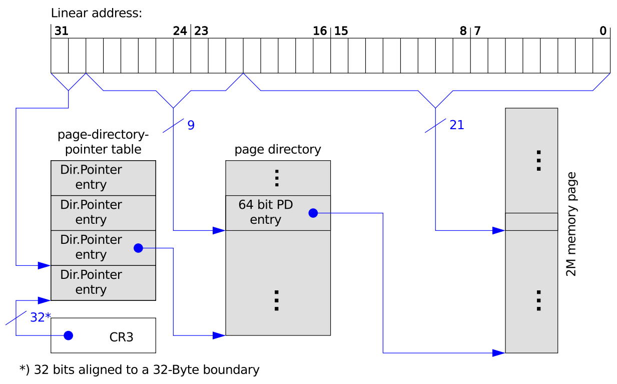
PDPT Page Directory Pointer Table, is a new level of Page Table
`64-bit entry` $\times 4$



PAE with 4K pages



PAE with 2M pages



Physical Address Extension (PAE)

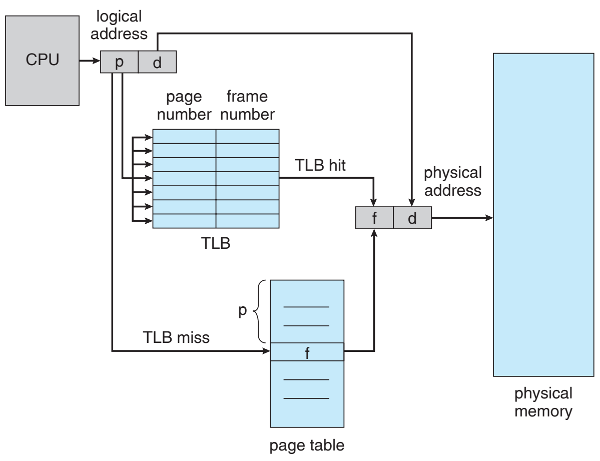
The linear address are still 32 bits

- A process cannot use more than 4G RAM
- The kernel programmers have to reuse the same linear addresses to map 64GB RAM
- The number of processes is increased

Translation Lookaside Buffers (TLB)

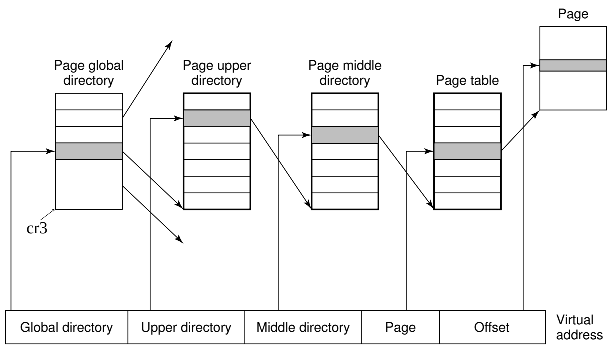
Fact: 80-20 rule

- Only a small fraction of the PTEs are heavily read; the rest are barely used at all



4.5 Paging in Linux

Paging In Linux
4-level paging for both 32-bit and 64-bit



4-level paging for both 32-bit and 64-bit

- 64-bit: four-level paging

  1. Page Global Directory
  2. Page Upper Directory
  3. Page Middle Directory
  4. Page Table

- 32-bit: two-level paging

  1. Page Global Directory
  2. Page Upper Directory — 0 bits; 1 entry
  3. Page Middle Directory — 0 bits; 1 entry
  4. Page Table

The same code can work on 32-bit and 64-bit architectures

Arch	Page size	Address bits	Paging levels	Address splitting
x86	4KB(12bits)	32	2	$10 + 0 + 0 + 10 + 12$
x86-PAE	4KB(12bits)	32	3	$2 + 0 + 9 + 9 + 12$
x86-64	4KB(12bits)	48	4	$9 + 9 + 9 + 9 + 12$

For 32-bit architectures with no PAE, two paging levels are sufficient. Linux essentially eliminates the PUD and the PMD fields by saying that they contain zero bits. However, *the positions of the PUD and the PMD in the sequence of pointers are kept so that the same code can work on 32-bit and 64-bit architectures*. [*Understanding The Linux Kernel*, Sec. 2.5, *Paging In Linux*]

Where are *the sequence of pointers* kept? The kernel keeps a position for the PUD and the PMD by setting the number of entries in them to 1 and mapping these two entries into the proper entry of the Page Global Directory.

How can Linux use 4-level model, while Intel specifies 2-level paging? (32-bit no PAE) Quoted from a stackoverflow page[33]:

> What I was trying to say was that the paging model used by the OS and the model used by the hardware are often distinct concepts. Linux uses a uniform paging model internally but this is layered on top of the hardware's paging model and requires architecture specific hacks to get it to work. It is the hardware's model that determines how address translation actually occurs (since it is the MMU that does this). *Linux simply adds a layer of indirection on top. Deep in its bowels it still uses the 10:10:12 model. — Abhay Buch Jul 8 '11 at 22:58*

More about paging Quoted from `http://linux-mm.org/VirtualMemory`:

> After bootup, all memory is accessed through the MMU which is paging enabled. So everything before and after `PAGE_OFFSET` is paged. Not everything can be demand paged, though ... (since kernel data structures are resident). Pages below `PAGE_OFFSET` belong to user space, and can be demand paged. Addresses above `PAGE_OFFSET` are kernel memory. There is a linear mapping for the first 900 MB of kernel memory, where physical address 0 ~ 896MB is mapped into `PAGE_OFFSET` ~ (`PAGE_OFFSET`

[33]`http://stackoverflow.com/questions/6627833/process-page-tables`

+ 896MB). So there are 896M/4K physical frames addressable from PAGE_OFFSET ~ (PAGE_OFFSET + 896MB).

Memory above PAGE_OFFSET is kernel virtual memory. Part of it is a direct map of the first part of physical memory. *but that same physical memory could also get virtual mappings from elsewhere*, eg. userspace or vmalloc. Also, userspace and vmalloc can map physical memory from outside the 896MB of direct mapped memory (as well as inside it).

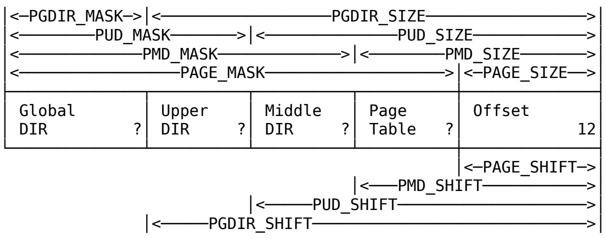
Most kernel memory allocation needs to come from that 896 MB, indeed, though page tables are the big exception ;) which means they're resident in memory all the time. Kernel data structures are always resident.

Process page tables could be either inside the low 896MB, or in highmem (or some page tables in both - more likely).

Physical memory is, by definition, not pageable. The contents of those pages might be pageable though. So you could have a page P at physical address 400MB. A process (eg. mozilla) is using that page at virtual address 120MB somewhere in its heap. The contents of the physical page can be paged out, at which point mozilla's heap page at 120MB is paged out. But the kernel mapping (at PAGE_OFFSET + 400MB) still maps the same page P just with different contents ;)

4.5.1 The Linear Address Fields

The Linear Address Fields



***_SHIFT** to specify the number of bits being mapped
***_MASK** to mask out all the upper bits
***_SIZE** how many bytes are addressed by each entry
*_MASK and *_SIZE values are calculated based on *_SHIFT
PAGE_SHIFT: Specifies the size of the area that a *page table entry* can cover
PMD_SHIFT: Specifies the size of the area that a *PMD entry* can cover

- When PAE is disabled, PMD_SHIFT yields the value 22 (12 from Offset plus 10 from Table)
- when PAE is enabled, PMD_SHIFT yields the value 21 (12 from Offset plus 9 from Table)
- LARGE_PAGE_SIZE = PMD_SIZE
  - 2^{22}, without PAE
  - 2^{21}, with PAE

PUD_SHIFT: Specifies the size of the area that a *PUD entry* can cover

- On the 80x86 processors, PUD_SHIFT is always equal to PMD_SHIFT
  - because both PUD and PMD are 0-bit long
- PUD_SIZE is equal to 4MB or 2MB.

PGDIR_SHIFT: Specifies the size of the area that a *PGDIR entry* can cover

- When PAE is disabled, PGDIR_SHIFT = PUD_SHIFT = PMD_SHIFT = 22
- when PAE is enabled, PGDIR_SHIFT $= 9_{PMD} + 9_{PT} + 12_{Offset} = 30$

include/asm-i386/page.h

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#define PAGE_MASK       (~(PAGE_SIZE-1))

#define LARGE_PAGE_MASK (~(LARGE_PAGE_SIZE-1))
#define LARGE_PAGE_SIZE (1UL << PMD_SHIFT)
```

PAGE_SIZE: $2^{12} = 4k$
PAGE_MASK: 0xfffff000
LARGE_PAGE_SIZE: depends
PAE: $2^{21} = 2M$
no PAE: $2^{22} = 4M$

Compile Time Dual-mode

`include/asm-i386/pgtable.h`

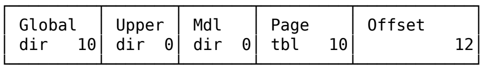
```
/*
 * The Linux x86 paging architecture is 'compile-time dual-mode', it
 * implements both the traditional 2-level x86 page tables and the
 * newer 3-level PAE-mode page tables.
 */
#ifdef CONFIG_X86_PAE
# include <asm/pgtable-3level_types.h>
# define PMD_SIZE        (1UL << PMD_SHIFT)
# define PMD_MASK        (~(PMD_SIZE - 1))
#else
# include <asm/pgtable-2level_types.h>
#endif

#define PGDIR_SIZE       (1UL << PGDIR_SHIFT)
#define PGDIR_MASK       (~(PGDIR_SIZE - 1))
```

	PMD_SHIFT	PUD_SHIFT	PGDIR_SHIFT
2-level	22	22	22
3-level	21	21	30

```
include/asm-i386/pgtable-2level-defs.h #define PGDIR_SHIFT        22
include/asm-i386/pgtable-3level-defs.h #define PGDIR_SHIFT        30
include/asm-x86_64/pgtable.h #define PGDIR_SHIFT  39
```

2-level — no PAE, 4K-page

PMD and PUD are folded



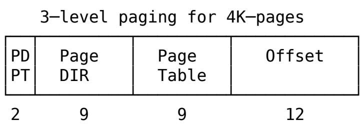
`include/asm-generic/pgtable-nopud.h`

```
#define PUD_SHIFT        PGDIR_SHIFT
#define PTRS_PER_PUD     1
#define PUD_SIZE         (1UL << PUD_SHIFT)
#define PUD_MASK         (~(PUD_SIZE-1))
```

`include/asm-generic/pgtable-nopmd.h`

```
#define PMD_SHIFT        PUD_SHIFT
#define PTRS_PER_PMD     1
#define PMD_SIZE         (1UL << PMD_SHIFT)
#define PMD_MASK         (~(PMD_SIZE-1))
```

3-level — PAE enabled



`include/asm-i386/pgtable-3level-defs.h`

```
#define PGDIR_SHIFT  30
#define PTRS_PER_PGD 4
#define PMD_SHIFT    21
#define PTRS_PER_PMD 512
```

PUD is eliminated

4-level — x86_64

48 address bits

Global DIR 9	Upper DIR 9	Middle DIR 9	Page Table 9	Offset 12

`include/asm-x86_64/pgtable.h`

```
#define PGDIR_SHIFT  39
#define PTRS_PER_PGD 512

#define PUD_SHIFT    30
#define PTRS_PER_PUD 512

#define PMD_SHIFT    21
#define PTRS_PER_PMD 512
```

4.5.2 Page Table Handling

Page Table Handling
— Data formats

`include/asm-i386/page.h`

```
#ifdef CONFIG_X86_PAE
extern unsigned long long __supported_pte_mask;
typedef struct { unsigned long pte_low, pte_high; } pte_t;
typedef struct { unsigned long long pmd; } pmd_t;
typedef struct { unsigned long long pgd; } pgd_t;
typedef struct { unsigned long long pgprot; } pgprot_t;
#define pmd_val(x)       ((x).pmd)
#define pte_val(x)       ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
#define __pmd(x) ((pmd_t) { (x) } )
#define HPAGE_SHIFT      21
#else
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
#define boot_pte_t pte_t /* or would you rather have a typedef */
#define pte_val(x)       ((x).pte_low)
#define HPAGE_SHIFT      22
#endif
```

Why structs? [*Professional Linux Kernel Architecture*, p158] `structs` are used instead of elementary types to ensure that the contents of page table elements are handled only by the associated helper functions and never directly.
[*Understanding the Linux Virtual Memory Manager*, Sec. 3.2, *Describing a Page Table Entry*] Even though these are often just unsigned integers, they are defined as structs for two reasons. The first is for type protection so that they will not be used inappropriately. The second is for features like PAE...

- For 32-bit systems,
  - long int: 4 bytes
  - long long int: 8 bytes
- `pgprot_t` holds the protection flags associated with a single entry.
  - Present/RW/User/Accessed/Dirty...
- `__pmd(x)`, type casting
- `pmd_val(x)`, reverse casting
- `HPAGE_SHIFT`, huge page shift.
  - 22 - without PAE, page size is $2^{22} = 4M$
  - 21 - with PAE, page size is $2^{21} = 2M$

Page Table Handling
Read or modify page table entries

Macros and functions

pte_none	pte_clear	set_pte	pte_same(a,b)
pte_present	pte_user()	pte_read()	pte_write()
pte_exec()	pte_dirty()	pte_young()	pte_file()
mk_pte_huge()	pte_wrprotect()	pte_rdprotect()	pte_exprotect()
pte_mkwrite()	pte_mkread()	pte_mkexec()	pte_mkclean()
pte_mkdirty()	pte_mkold()	pte_mkyoung()	pte_modify(p,v)
mk_pte(p,prot)	pte_index(addr)	pte_page(x)	pte_to_pgoff(pte)

a lot more for pmd, pud, pgd ...

Example — To find a page table entry
mm/memory.c

```
pgd_t *pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *ptep, pte;

pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
  goto out;

pud = pud_offset(pgd, address);
if (pud_none(*pud) || unlikely(pud_bad(*pud)))
  goto out;

pmd = pmd_offset(pud, address);
if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
  goto out;

ptep = pte_offset_map(pmd, address);
if (!ptep)
  goto out;

pte = *ptep;
```

`pgd_offset(mm, addr)` Receives as parameters the address of a memory descriptor `mm` [*Understanding The Linux Kernel*, Chapter 9, *Process Address Space*] and a linear address `addr`. The macro yields the linear address of the entry in a Page Global Directory that corresponds to the address `addr`; the Page Global Directory is found through a pointer within the memory descriptor.

Memory descriptor [*Understanding The Linux Kernel*, Sec 9.2, *The Memory Descriptor*] All information related to the process address space is included in an object called the memory descriptor of type `mm_struct`. This object is referenced by the `mm` field of the process descriptor.

Line 312 in `pgtable.h`:

```
/*
 * pgd_offset() returns a (pgd_t *)
 * pgd_index() is used get the offset into the pgd page's array of pgd_t's;
 */
#define pgd_offset(mm, address) ((mm)->pgd+pgd_index(address))
```

Line 305 in `pgtable.h`:

```
/*
 * the pgd page can be thought of an array like this: pgd_t[PTRS_PER_PGD]
 *
 * this macro returns the index of the entry in the pgd page which would
 * control the given virtual address
 */
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))
```

$$\mathtt{PTRS\_PER\_PGD} = \begin{cases} 1024 & \mathtt{i386, noPAE} \\ 4 & \mathtt{i386, PAE} \\ 512 & \mathtt{x86\_64} \end{cases}$$

`pud_offset(pgd, addr)` Receives as parameters a pointer `pgd` to a Page Global Directory entry and a linear address `addr`. The macro yields the linear address of the entry in a Page Upper Directory that corresponds to `addr`. In a two- or three-level paging system, this macro yields `pgd`, the address of a Page Global Directory entry.

Line 36 in `pgtable-nopud.h`:

```
static inline pud_t * pud_offset(pgd_t * pgd, unsigned long address)
{
        return (pud_t *)pgd;
}
```

`pmd_offset(pud, addr)` Receives as parameters a pointer `pud` to a Page Upper Directory entry and a linear address `addr`. The macro yields the address of the entry in a Page Middle Directory that corresponds to `addr`. In a two-level paging system, it yields `pud`, the address of a Page Global Directory entry.

Line 39 in `pgtable-nopmd.h`:

```
static inline pmd_t * pmd_offset(pud_t * pud, unsigned long address)
{
        return (pmd_t *)pud;
}
```

`pte_offset_kernel(dir, addr)` Yields the linear address of the Page Table that corresponds to the linear address `addr` mapped by the Page Middle Directory `dir`. Used only on the master kernel page tables (See also [*Understanding The Linux Kernel*, Sec. 2.5.5, *Kernel Page Tables*]).
Line 335 in `pgtable.h`:

```
/*
 * the pte page can be thought of an array like this: pte_t[PTRS_PER_PTE]
 *
 * this macro returns the index of the entry in the pte page which would
 * control the given virtual address
 */
#define pte_index(address) \
                (((address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset_kernel(dir, address) \
        ((pte_t *) pmd_page_kernel(*(dir)) +  pte_index(address))
```

`pte_offset_map(dir, addr)` Receives as parameters a pointer `dir` to a Page Middle Directory entry and a linear address `addr`; it yields the linear address of the entry in the Page Table that corresponds to the linear address `addr`. If the Page Table is kept in high memory, the kernel establishes a temporary kernel mapping (See also [*Understanding The Linux Kernel*, Sec. 8.1.6, *Kernel Mappings of High-Memory Page Frames*]), to be released by means of `pte_unmap`. The macros `pte_offset_map_nested` and `pte_unmap_nested` are identical, but they use a different temporary kernel mapping.
Line 370 in `pgtable.h`:

```
#if defined(CONFIG_HIGHPTE)
#define pte_offset_map(dir, address) \
        ((pte_t *)kmap_atomic(pmd_page(*(dir)),KM_PTE0) + pte_index(address))
#define pte_offset_map_nested(dir, address) \
        ((pte_t *)kmap_atomic(pmd_page(*(dir)),KM_PTE1) + pte_index(address))
#define pte_unmap(pte) kunmap_atomic(pte, KM_PTE0)
#define pte_unmap_nested(pte) kunmap_atomic(pte, KM_PTE1)
#else
#define pte_offset_map(dir, address) \
        ((pte_t *)page_address(pmd_page(*(dir))) + pte_index(address))
#define pte_offset_map_nested(dir, address) pte_offset_map(dir, address)
#define pte_unmap(pte) do { } while (0)
#define pte_unmap_nested(pte) do { } while (0)
#endif
```

`pte_none, pmd_none, pud_none, pgd_none` yield the value 1 if the corresponding entry has the value 0; otherwise, they yield the value 0.

`pmd_bad, pud_bad, pgd_bad` The `pmd_bad` macro is used by functions to check Page Middle Directory entries passed as input parameters. It yields the value 1 if the entry points to a bad Page Table that is, if at least one of the following conditions applies:
- The page is not in main memory (Present flag cleared).
- The page allows only Read access (Read/Write flag cleared).
- Either Accessed or Dirty is cleared (Linux always forces these flags to be set for every existing Page Table).

The `pud_bad` and `pgd_bad` macros always yield 0. No `pte_bad` macro is defined, because it is legal for a Page Table entry to refer to a page that is not present in main memory, not writable, or not accessible at all.

4.5.3 Physical Memory Layout

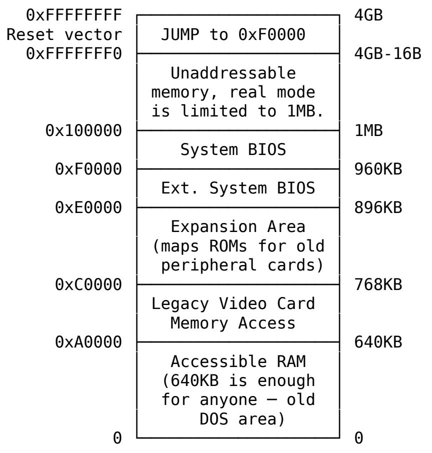
Physical Memory Layout
`0x00100000` — The kernel starting point

Reserved page frames

- unavailable to users
- kernel code and data structures
- no dynamic assignment, no swap out

The kernel is loaded starting from the second megabyte (`0x00100000`) in RAM

- Page frame 0 — BIOS
- $640K \sim 1M$ — the well-know hole
- `/proc/iomem`


0xFFFFFFFF
Reset vector
0xFFFFFFF0
JUMP to 0xF0000
4GB
4GB-16B
Unaddressable memory, real mode is limited to 1MB.
0x100000
1MB
System BIOS
0xF0000
960KB
Ext. System BIOS
0xE0000
896KB
Expansion Area (maps ROMs for old peripheral cards)
0xC0000
768KB
Legacy Video Card Memory Access
0xA0000
640KB
Accessible RAM (640KB is enough for anyone - old DOS area)
0
0

Why isn't the kernel loaded starting with the first available megabyte of RAM? [*Understanding The Linux Kernel*, Sec. 2.5.3, *inline*] Well, the PC architecture has several peculiarities that must be taken into account. For example:

- Page frame 0 is used by BIOS to store the system hardware configuration detected during the Power-On Self-Test(POST); the BIOS of many laptops, moreover, writes data on this page frame even after the system is initialized.
- Physical addresses ranging from `0x000a0000` to `0x000fffff` are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards. This area is the well-known hole from 640 KB to 1 MB in all IBM-compatible PCs: the physical addresses exist but they are reserved, and the corresponding page frames cannot be used by the operating system.
- Additional page frames within the first megabyte may be reserved by specific computer models. For example, the IBM ThinkPad maps the `0xa0` page frame into the `0x9f` one.

While booting

1. The kernel queries the BIOS for available physical address ranges
2. `machine_specific_memory_setup()` — builds the physical addresses map
3. `setup_memory()` — initializes a few variables that describe the kernel's physical memory layout
   - `min_low_pfn, max_low_pfn, highstart_pfn, highend_pfn, max_pfn`

See also: [*Professional Linux Kernel Architecture*, Sec. 3.4, *Memory Initialization Steps*]

```
setup_arch
├─ machine_specific_memory_setup
├─ parse_early_param
├─ setup_memory
├─ paging_init
│  └─ pagetable_init
└─ zone_sizes_init
   ├─ add_active_range
   └─ free_area_init_nodes
```

- `setup_arch` is invoked from within `start_kernel()`
- `machine_specific_memory_setup`: to create a list with the memory regions occupied by the system and the free memory regions
- `parse_early_param`: parsing commandline arguments like `mem=XXX[KkmM]`, `highmem=XXX[kKmM]`, or `memmap=XXX[KkmM]""@XXX[KkmM]` arguments
- `setup_memory`
  - The number of physical pages available (per node) is determined.
  - The bootmem allocator is initialized [*Professional Linux Kernel Architecture*, Sec. 3.4.3, *Memory Management During The Boot Process*]
  - Various memory areas are then reserved, for instance, for the initial RAM disk needed when running the first userspace processes.
- `paging_init`: initializes the kernel page tables and enables paging
- `pagetable_init`: initializes the direct mapping of physical memory into the kernel address space. All page frames in low memory are directly mapped to the virtual memory region above `PAGE_OFFSET`. *This allows the kernel to address a good part of the available memory without having to deal with page tables anymore.*
- `zone_sizes_init`: initializes the `pgdat_t` instances of all nodes of the system
  1. `add_active_range`: a comparatively simple list of the available physical memory is prepared
  2. `free_area_init_nodes`: uses this information (got in above step) to prepare the full-blown kernel data structures

BIOS-Provided Physical Addresses Map

Example — a typical computer with 128MB RAM

Start	End	Type
0x00000000	0x0009ffff (640K)	Usable
0x000f0000 (960K)	0x000fffff (1M-1)	Reserved
0x00100000 (1M)	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff (128M)	ACPI NVS
0xffff0000	0xffffffff	Reserved

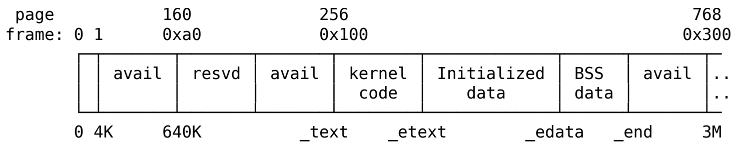
- The *ACPI data* area stores information about the hardware devices of the system written by the BIOS in the POST phase; during the initialization phase, the kernel copies such information in a suitable kernel data structure, and then considers these page frames usable.

- The *ACPI NVS* area is mapped to ROM chips of the hardware devices, and hence cannot be used.
- The physical address range starting from `0xffff0000` is marked as reserved, because it is mapped by the hardware to the BIOS's ROM chip [*Understanding The Linux Kernel*, *Appendix A*]
- Notice that the BIOS may not provide information for some physical address ranges (in the table, the range is `0x000a0000` to `0x000effff`). To be on the safe side, Linux assumes that such ranges are not usable.
- The kernel might not see all physical memory reported by the BIOS: for instance, the kernel can address only 4GB of RAM if it has not been compiled with PAE support, even if a larger amount of physical memory is actually available.
- kernlediy.com: Boot Memory Allocator
- [*Professional Linux Kernel Architecture*, Sec. 3.4.3, *Memory Management During The Boot Process*]

Variables describing the physical memory layout

Variable name	Description
num_physpages	Page frame number of the highest usable page frame
totalram_pages	Total number of usable page frames
min_low_pfn	Page frame number of the first usable page frame after the kernel image in RAM
max_pfn	Page frame number of the last usable page frame
max_low_pfn	Page frame number of the last page frame directly mapped by the kernel (low memory)
totalhigh_pages	Total number of page frames not directly mapped by the kernel (high memory)
highstart_pfn	Page frame number of the first page frame not directly mapped by the kernel
highend_pfn	Page frame number of the last page frame not directly mapped by the kernel

The first 768 page frames (3 MB) in Linux 2.6



- You can find the linear address of these symbols (`_text, _etext, _edata, _end`) in the file `System.map`, which is created right after the kernel is compiled.

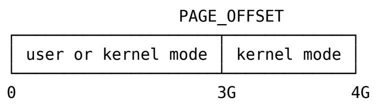
4.5.4 Process Page Tables

Process Page Tables

`0xC0000000` ⇔ `PAGE_OFFSET`

`include/asm-i386/page.h`

```
#define __PAGE_OFFSET           (0xC0000000)
#define PAGE_OFFSET             ((unsigned long)__PAGE_OFFSET)
```



Why?

- easy to switch to kernel mode
- easy physical addressing due to direct mapping

$$Physical = Virtual - \texttt{PAGE\_OFFSET}$$

4G/4G solution

- (LWN article) There are users out there wanting to scale 32-bit Linux systems up to 32GB or more of main memory, so the enterprise-oriented Linux distributors have been scrambling to make that possible. One approach is the 4G/4G patch written by Ingo Molnar. This patch separates the kernel and user address spaces, allowing user processes to have 4GB of virtual memory while simultaneously expanding the kernel's low memory to 4GB. There is a cost, however: *the translation buffer (TLB) is no longer shared and must be flushed for every transition between kernel and user space*. Estimates of the magnitude of the performance hit vary greatly, but numbers as high as 30% have been thrown around. This option makes some systems work, however, so Red Hat ships a 4G/4G kernel with its enterprise offerings.
  Better solution: **go get a 64-bit system**.

- (LWN article: 4G/4G split on x86, 64 GB RAM (and more) support) Performance impact of the 4G/4G feature:
  There's a runtime cost with the 4G/4G patch: to implement separate address spaces for the kernel and userspace VM, the entry/exit code has to switch between the kernel pagetables and the user pagetables. This causes TLB flushes, which are quite expensive, not so much in terms of TLB misses (which are quite fast on Intel CPUs if they come from caches), but in terms of the direct TLB flushing cost (`cr3` manipulation) done on system-entry.
- It would also be possible to get rid of the split completely by introducing two 4 GiB address spaces, one for the kernel and one for each userspace program. However, context switches between kernel and user mode are more costly in this case. [*Professional Linux Kernel Architecture*, Sec 3.4.2,*Initialization of Paging*, p175]

Linux Memory Management Overview (a bit old)[tldp.org]

- A process' `PGDir` is initialized during a fork by `copy_page_tables()`. The idle process (`swapper`) has its `PGDir` initialized during the initialization sequence (`swapper_pg_dir`).
- The kernel code and data segments are priveleged segments defined in the global descriptor table (GDT) and extend from 3 GB to 4 GB. The swapper page directory (`swapper_pg_dir`) is set up so that logical addresses and physical addresses are identical in kernel space.
- Each user process has a local descriptor table (LDT) that contains a code segment and data-stack segment. These user segments extend from 0 to 3 GB (`0xc0000000`). In user space, linear addresses and logical addresses are identical.
- The space above 3 GB appears in a process' `PGDir` as pointers (each entry in `PGDir` has a pointer) to kernel page tables. [*Understanding The Linux Kernel*, Sec 2.5.4, *The entries of the PGDir*]
  This space is invisible to the process in user mode but the mapping becomes relevant when privileged mode is entered, for example, to handle a system call. Supervisor mode is entered within the context of the current process so address translation occurs with respect to the process' `PGDir` but using kernel segments. This is identically the mapping produced by using the `swapper_pg_dir` and kernel segments as both page directories use the same page tables in this space. Only `task[0]` (the idle task, sometimes called the swapper task for historical reasons, even though it has nothing to do with swapping in the Linux implementation) uses the `swapper_pg_dir` directly.
  - The user process' `segment_base = 0x00`, `page_dir` private to the process.
  - user process makes a system call: `segment_base = 0xc0000000`, `page_dir = same user page_dir`.
  - `swapper_pg_dir` contains a mapping for all physical pages from `0xc0000000` to `0xc0000000 + end_mem`, so the first 768 entries in `swapper_pg_dir` are 0's, and then there are 4 or more that point to kernel page tables.
  - The user page directories have the same entries as `swapper_pg_dir` above 768. The first 768 entries map the user space.
- The upshot is that whenever the linear address is above `0xc0000000` everything uses the same kernel page tables.
- The user stack sits at the top of the user data segment and grows down. The kernel stack is not a pretty data structure or segment that I can point to with a "yon lies the kernel stack." A `kernel_stack_frame` (a page) is associated with each newly created process and is used whenever the kernel operates within the context of that process. Bad things would happen if the kernel stack were to grow below its current stack frame.

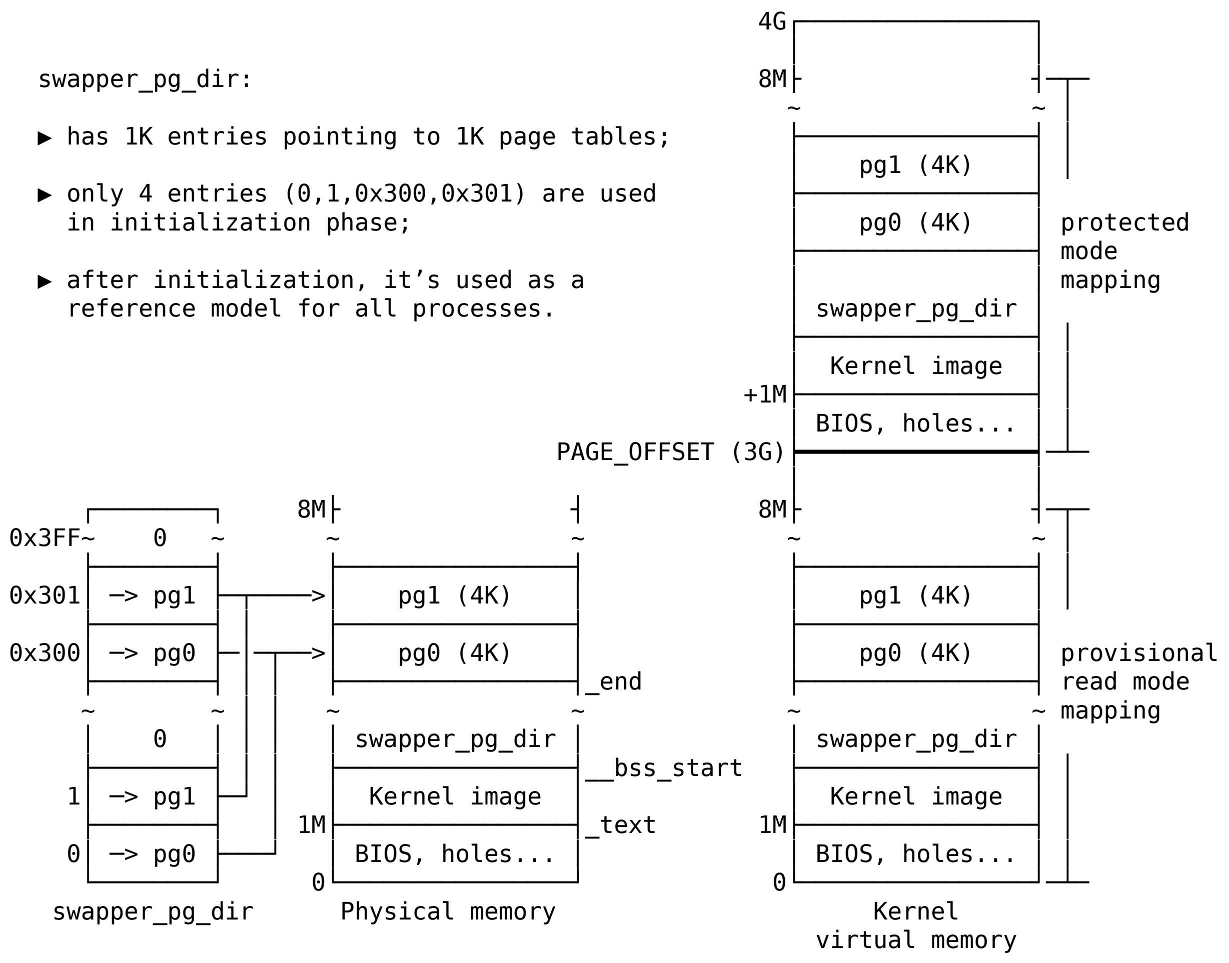
The entries of the PGDir [*Understanding The Linux Kernel*, Sec. 2.5.4]

- lower than `0xc0000000`: (the first 768 entries with PAE disabled, or the first 3 entries with PAE enabled) depends on the specific process
  - Each user process thinks it has 3 GiB of memory
- higher than `0xc0000000`: can be addressed only when the process is in kernel mode. This address space is common to all the processes and equal to the corresponding entries of the master kernel PGDir (see the following section)
- For 32-bit systems without PAE, `PGDir`
  - is 10-bit long
  - has 2^{10} (1K) entries
  - each PGDIR entry covers 2^{22} (4M)
  - the first 768 entries cover $768 \times 2^{22} = 3G$
- For 32-bit systems with PAE enabled, `PGDir`
  - is 2-bit long
  - has $2^2 = 4$ entries
  - each `PGDIR` entry covers $2^{30} = 1G$
  - the first 3 entries cover $3 \times 2^{30} = 3G$
- The address space after `PAGE_OFFSET` is reserved for the kernel and this is where the complete physical memory is mapped (eg. if a system has 64mb of RAM, it is mapped from `PAGE_OFFSET` to `PAGE_OFFSET + 64mb`). This address space is also used to map non-continuous physical memory into continuous virtual mem-

ory. [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.3.1, *Significance of PAGE_OFFSET*]

4.5.5 Kernel Page Tables

Kernel Page Tables
Master Kernel Page Global Directory



swapper_pg_dir:

- ▶ has 1K entries pointing to 1K page tables;
- ▶ only 4 entries (0,1,0x300,0x301) are used in initialization phase;
- ▶ after initialization, it's used as a reference model for all processes.

In The Beginning, There Is No Paging

Before tuning on paging, the page tables must be ready
Two phases:

1. **Bootstrapping:** sets up page tables for just 8MB so the paging unit can be enabled
   8MB? 2 page tables (pg0, pg1), enough to handle the kernel's code and data segments, and 128 KB for some dynamic data structures (page frame bitmap)
2. **Finalising:** initializes the rest of the page tables

Provisional Page Global Directory

- A provisional PGDir is initialized statically during kernel compilation

```
.section ".bss.page_aligned","w"
ENTRY(swapper_pg_dir)
        .fill 1024,4,0
```

- The provisional PTs are initialized by `startup_32()` in `arch/i386/kernel/head.S`
- `swapper_pg_dir` — A 4KB area for holding provisional PGDir
- provisional PGDir has only 4 useful entries: `0, 1, 0x300, 0x301`

What's it for?

Linear		Physical
$0 \sim 8MB$	$\Rightarrow$	$0 \sim 8MB$
$\texttt{PAGE\_OFFSET} \sim (\texttt{PAGE\_OFFSET} + 8MB)$	$\nearrow$	

So that the kernel image ($< 8MB$) in physical memory can be addressed in both real mode and protected mode.

- "`.fill 1024, 4, 0`" initializes a 4K area for `swapper_pg_dir`[34].
- `swapper_pg_dir` is at the beginning of BSS (uninitialized data area) because BSS is no longer used after system start up.
- assuming that the kernel's segments, the provisional Page Tables, and the 128KB memory area fit in the first 8 MB of RAM.
- In order to map 8 MB of RAM, two Page Tables are required.
- `pg0` is right after the end of BSS (`_end`).

We won't bother mentioning the Page Upper Directories and Page Middle Directories anymore, because they are equated to Page Global Directory entries. PAE support is not enabled at this stage.

The objective of this first phase of paging is *to allow these 8 MB of physical RAM to be easily addressed both in real mode and protected mode*. Therefore, the kernel must create a mapping from both the linear addresses `0x00000000` through `0x007fffff` (8M) and the linear addresses `0xc0000000` through `0xc07fffff` (8M) into the physical addresses `0x00000000` through `0x007fffff`. In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from `0xc0000000`.

Why? Quoted from [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec 1.3.2]

- All pointers in the compiled kernel refer to addresses `> PAGE_OFFSET`. That is, the kernel is linked under the assumption that its base address will be `start_text` (I think; I don't have the code on hand at the moment), which is defined to be `PAGE_OFFSET`+(some small constant, call it `C`).
- All the kernel bootstrap code (mostly real mode code) is linked assuming that its base address is `0+C`.

`head.S` is part of the bootstrap code. It's running in protected mode with paging turned off, so all addresses are physical. In particular, the instruction pointer is fetching instructions based on physical address. The instruction that turns on paging (`movl %eax, %cr0`) is located, say, at some physical address `A`.

As soon as we set the paging bit in `cr0`, paging is enabled, and starting at the very next instruction, all addressing, including instruction fetches, pass through the address translation mechanism (page tables), IOW, all addresses are henceforth virtual. That means that

1. We must have valid page tables, and
2. Those tables must properly map the instruction pointer to the next instruction to be executed.

That next instruction is physically located at address `A+4` (the address immediately after the "`movl %eax, %cr0`" instruction), but from the point of view of all the kernel code — which has been linked at `PAGE_OFFSET` — that instruction is located at virtual address `PAGE_OFFSET+(A+4)`. Turning on paging, however, does not magically change the value of EIP (The value of EIP is still physically `A+4`, not `PAGE_OFFSET+(A+4)` yet. But since paging is just enabled, CPU could pass `A+4` through address translation). The CPU fetches the next instruction from *virtual* address `A+4`; that instruction is the beginning of a short sequence that effectively relocates the instruction pointer to point to the code at `PAGE_OFFSET+A+(something)`.

But since the CPU is, for those few instructions, fetching instructions based on physical addresses but *having those instructions pass through address translation*, we must ensure that both the physical addresses and the virtual addresses are :

1. Valid virtual addresses, and
2. Point to the same code.

That means that at the very least, the initial page tables must

1. map virtual address `PAGE_OFFSET+(A+4)` to physical address `(A+4)`, and must
2. map virtual address `A+4` to physical address `A+4`.

This dual mapping for the first 8MB of physical RAM is exactly what the initial page tables accomplish. The 8MB initally mapped is more or less arbitrary. It's certain that no bootable kernel will be greater than 8MB in size. The identity mapping is discarded when the MM system gets initialized.

Provisional Page Table Initialization

`arch/i386/kernel/head.S`

[34] `.fill REPEAT, SIZE, VALUE`: This emits `REPEAT` copies of `SIZE` bytes.

```
page_pde_offset = (__PAGE_OFFSET >> 20);

        movl $(pg0 - __PAGE_OFFSET), %edi
        movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
        movl $0x007, %eax       # 0x007 = PRESENT+RW+USER
10:
        leal 0x007(%edi), %ecx  # Create PDE entry
        movl %ecx, (%edx)       # Store identity PDE entry
        movl %ecx, page_pde_offset(%edx)   # Store kernel PDE entry
        addl $4, %edx
        movl $1024, %ecx
11:
        stosl    # movl %eax, (%edi)
                 # addl $4, %edi
        addl $0x1000, %eax
        loop 11b
        # End condition: we must map up to and including INIT_MAP_BEYOND_END
        # bytes beyond the end of our own page tables; the +0x007 is the
        # attribute bits
        leal (INIT_MAP_BEYOND_END + 0x007)(%edi), %ebp
        cmpl %ebp, %eax
        jb 10b
        movl %edi, (init_pg_tables_end - __PAGE_OFFSET)
```

- The identity mapping is discarded when the MM system gets initialized.
- `page_pde_offset = (__PAGE_OFFSET >> 20);` */* 0xC00, the 3K point.*/*
  The `PGDir` is one page (4K) in size. It's divided into two parts:
  1. first 3K (768 entries) for user mode
  2. last 1K (256 entries) for kernel mode
- "`$(pg0 - __PAGE_OFFSET)`" yields the physical address of `pg0` since here it is a linear address. Same case for "`$(swapper_pg_dir - __PAGE_OFFSET)`"
- Registers:
  `%edi` address of each page table entry, i.e. `pg0[0]..pg0[1023]`, `pg1[0]..pg1[1023]`.
  `%edx` address of `swapper_pg_dir[0]` and then to `swapper_pg_dir[1]`.
  `%ecx` has two uses
    1. contents of `swapper_pg_dir[0]`, `swapper_pg_dir[1]`, `swapper_pg_dir[768]`, `swapper_pg_dir[769]`.
    2. loop counter (1024 -> 0)

  `%eax 7, 4k+7, 8k+7 ... 8M-4k+7` for 2k page table entries in `pg0` and `pg1` respectively.
  `%ebp = 128k + 7 + &pg0[1023]` in the first round of loop. Its value cannot be determined at coding time, because the address of `pg0` is not known until compile/link time.
- `stosl`: stores the contents of EAX at the address pointed by `EDI`, and increments `EDI`. Equivalent to:

```
movl %eax, (%edi)
addl $4, %edi
```

- `cmpl, jb`: if `%eax` < `%ebp`, jump to 10;
  - At the end of the 1st round of loop, the value of `%eax` is `4M-4k+7`, while the value of `%ebp` depends on the address of `pg0`.
- `INIT_MAP_BEYOND_END`: 128KB[35]

Equivalent pseudo C code

```
/*
 * Provisional PGDir and page tables setup
 *
 * for mapping two linear address ranges to the same physical address range
 *
 *   + Linear address ranges:
 *              -   User mode: $i\times{}4M\sim{}(i+1)\times{}4M-1$
 *              - Kernel mode: $3G+i\times{}4M\sim{}3G+(i+1)\times{}4M-1$
 *   + Physical address range: $i\times{}4M\sim{}(i+1)\times{}4M-1$
 */
typedef unsigned int PTE;
PTE *pg = pg0;      /* physical address of pg0 */
PTE pte = 0x007;    /* 0x007 = PRESENT+RW+USER */
for(i=0;;i++){
  swapper_pg_dir[i] = pg + 0x007;          /* store identity PDE entry */
  swapper_pg_dir[i+page_pde_offset] = pg + 0x007; /* kernel PDE entry */
  for(j=0;j<1024;j++){                /* populating one page table */
    pg[i*1024 + j] = pte;             /* fill up one page table entry */
    pte += 0x1000;                    /* next 4k */
  }
  if(pte >= ((char*)pg + i*1024 + j)*4 + 0x007 + INIT_MAP_BEYOND_END)
    {
      init_pg_tables_end = pg + i*0x1000 + j;
      break;
    }
  }
```

[35] dead link: `http://kerneldiy.com/blog/?p=201`

See also: `http://www.eefocus.com/article/09-04/71517s.html`

Enable paging

`startup_32()` in `arch/i386/kernel/head.S`

```
# Enable paging
movl $swapper_pg_dir - __PAGE_OFFSET, %eax
movl %eax, %cr3   # set the page table pointer..
movl %cr0, %eax
 orl $0x80000000, %eax
movl %eax, %cr0   # ..and set paging (PG) bit
```

Final Kernel Page Table Setup

- `master kernel PGDir` is still in `swapper_pg_dir`
- initialized by `paging_init()`

Situations

1. `RAM size` $< 896M$
   - every RAM cell is mapped
2. $896M <$ `RAM size` $< 4G$
   - 896M are mapped
3. `RAM size` $> 4G$
   - PAE enabled

When RAM size is less than 896 MB

`paging_init()` without PAE

```
void __init paging_init(void)
{
#ifdef CONFIG_X86_PAE
  /* ... */
#endif

  pagetable_init();
  load_cr3(swapper_pg_dir);

#ifdef CONFIG_X86_PAE
  /* ... */
#endif

  __flush_tlb_all();
  kmap_init();
  zone_sizes_init();
}
```

Is PAE enabled in your kernel? try "`grep PAE /boot/config-*`"

- `paging_init()` without PAE:
  1. Invokes `pagetable_init()` to set up the Page Table entries properly.
  2. Writes the physical address of `swapper_pg_dir` in the cr3 control register.
  3. Invokes `__flush_tlb_all()` to invalidate all TLB entries.

2 level paging: PUD and PMD are folded

Global dir 10	Upper dir 0	Mdl dir 0	Page tbl 10	Offset 12

`pagetable_init()` — re-initializes the `PGDir` at `swapper_pg_dir`

Equivalent code:

```
pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
phys_addr = 0x00000000;
while (phys_addr < (max_low_pfn * PAGE_SIZE))
{
  pmd = one_md_table_init(pgd); /* returns pgd itself */
  set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
  /* 0x1e3 == Present, Accessed, Dirty, Read/Write, Page Size, Global */
  phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x400000, 4M */
  ++pgd;
}
```

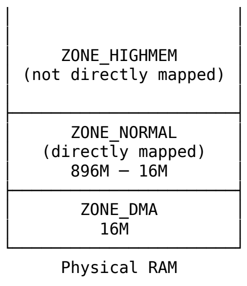
- [*Understanding The Linux Kernel*, Sec. 2.5.5.2, *Final kernel Page Table when RAM size is less than 896 MB*]

- This loop begins setting up valid PMD entries to point to. In the PAE case, pages are allocated with `alloc_bootmem_low_page` and the PGD is set appropriately. Without PAE, there is no middle directory, so it is just "folded"back onto the PGD to preserve the illusion of a 3-level pagetable. [*Understanding the Linux Virtual Memory Manager*, Appendix C, *Page Table Management*, p224]
- `#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))`
  - `(PAGE_OFFSET >> PGDIR_SHIFT) & (PTRS_PER_PGD-1)` ⇒
  - `(C0000000 >> 22) & (1024 - 1) = 0x300 & 1023 = 768`
- pgd is a pointer initialized to the `pgd_t` corresponding to the beginning of the kernel portion of the linear address space. The lower 768 entries are left alone for user space.
- `one_md_table_init()` is described in the comments of the source code. And it's easy to trace the calls in it to get a clear idea.
  Line 55 in `mm/init.c`:

```c
/*
 * Creates a middle page table and puts a pointer to it in the
 * given global directory entry. This only returns the gd entry
 * in non-PAE compilation mode, since the middle layer is folded.
 */
static pmd_t * __init one_md_table_init(pgd_t *pgd)
{
  pud_t *pud;
  pmd_t *pmd_table;

#ifdef CONFIG_X86_PAE
  ...
#else
  pud = pud_offset(pgd, 0);
  pmd_table = pmd_offset(pud, 0);
#endif

  return pmd_table;
}
```

- `#define set_pmd(pmdptr, pmdval) (*(pmdptr) = (pmdval))`

When RAM Size Is Between 896MB ∼ 4096MB
Physical memory zones:

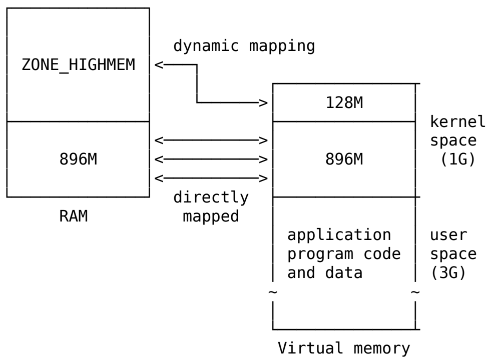


Direct mapping for `ZONE_NORMAL`:

```c
#define __pa(x)  ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)  ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

- To initialize the Page Global Directory, the kernel uses the same code as in the previous case.
- The `__pa` macro is used to convert a linear address starting from `PAGE_OFFSET` to the corresponding physical address, while the `__va` macro does the reverse.
- `ZONE_DMA` - Contains page frames of memory below 16 MB
- `ZONE_NORMAL` - Contains page frames of memory at and above 16 MB and below 896 MB
- `ZONE_HIGHMEM` - Contains page frames of memory at and above 896 MB

High Memory



- Coping with HighMemory [*HighMemory*]
  - Memory above the physical address of 896MB are temporarily mapped into kernel virtual memory whenever the kernel needs to access that memory.
  - Data which the kernel frequently needs to access is allocated in the lower 896MB of memory (`ZONE_NORMAL`) and can be immediately accessed by the kernel (see Temporary mapping).
  - Data which the kernel only needs to access occasionally, including page cache, process memory and page tables, are preferentially allocated from `ZONE_HIGHMEM`.
  - The system can have additional physical memory zones to deal with devices that can only perform DMA to a limited amount of physical memory, `ZONE_DMA` and `ZONE_DMA32`.
  - Allocations and pageout pressure on the various memory zones need to be balanced (see Memory Balancing).
- (From comments in http://kerneltrap.org/node/2450) **The whole 1st GB of physical RAM is reserved for kernel use?** In fact, it is. The kernel has to have control over the whole memory. It's the kernel's job to allocate/deallocate RAM to processes.
  What the kernel does is, maps the entire available memory in its own space, so that it can access any memory region. It then gives out free pages to processes which need them.
  The userspace processes cannot of its own allocate a page to itself. It has to request the kernel to give it some memory area. Once the kernel has mapped a physical page in the process's space, it can use that extra memory.
- To access physical memory between the range of 1GiB and 4GiB, the kernel temporarily maps pages from high memory into `ZONE_NORMAL` with `kmap()`. [*Understanding the Linux Virtual Memory Manager*, Sec. 2.5, *High Memory*]
  Why? [*Understanding The Linux Kernel*, Sec 8.1.6, *Kernel Mappings of High-Memory Page Frames*] Page frames above the 896 MB boundary are not generally mapped in the 4th GiB of the kernel linear address spaces, so the kernel is unable to directly access them. This implies that each page allocator function that returns the linear address of the assigned page frame doesn't work for high-memory page frames, that is, for page frames in the `ZONE_HIGHMEM` memory zone.
  For instance, suppose that the kernel invoked `__get_free_pages(GFP_HIGHMEM,0)` to allocate a page frame in high memory. If the allocator assigned a page frame in high memory, `__get_free_pages()` cannot return its linear address because it doesn't exist (there is no mapping between physical frame and virtual page); thus, the function returns NULL. In turn, the kernel cannot use the page frame; even worse, the page frame cannot be released because the kernel has lost track of it.
  Linux designers had to find some way to allow the kernel to exploit all the available RAM, up to the 64 GB supported by PAE. The approach adopted is the following:
  - The allocation of high-memory page frames is done only through the `alloc_pages()` function and its `alloc_page()` shortcut. These functions do not return the linear address of the first allocated page frame, because if the page frame belongs to the high memory, such linear address simply does not exist. Instead, *the functions return the linear address of the page descriptor of the first allocated page frame* [*Understanding The Linux Kernel*, Sec 8.1.1, *Page Descriptors*]. These linear addresses always exist, because all page descriptors are allocated in low memory once and forever during the kernel initialization.
  - Page frames in high memory that do not have a linear address cannot be accessed by the kernel. Therefore, part of the last 128 MB of the kernel linear address space is dedicated to mapping high-memory page frames. Of course, this kind of mapping is temporary, otherwise only 128 MB of high memory would be accessible. Instead, by recycling linear addresses the whole high memory can be accessed, although at different times.
- More about high memory:
  - [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.8, *Page Table Setup*]

  – Stackoverflow: How does the linux kernel manage less than 1GB physical memory?
  – `http://www.cs.usfca.edu/~cruse/cs635/lesson04.ppt`
  – `http://ilinuxkernel.com/?p=1013`

When RAM Size Is More Than 4096MB (PAE)

A 3-level paging model is used

```
           3-level paging for 4K-pages
       ┌──┬─────────┬─────────┬───────────┐
       │PD│  Page   │  Page   │  Offset   │
       │PT│  DIR    │  Table  │           │
       └──┴─────────┴─────────┴───────────┘
        2      9         9          12
```

PGDir	PUD	PMD	PT	OFFSET
2	0	9	9	12

The PGDir is initialized by a cycle equivalent to the following:

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */

/* the first 3 entries are for user space, and are pointing to the same empty_zero_page.*/
for (i=0; i<pgd_idx; i++)
  set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001)); /* 0x001 == Present */

/* the 4th entry is for kernel space*/
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;

/* i=3 initially. PTRS_PER_PGD=4 */
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
  /* get the address of a PMD.
     The PMD maps 1G allocated by alloc_bootmem_low_pages() */
  pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
  /* the 4th entry is initialized with the above PMD */
  set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */

  if (phys_addr < max_low_pfn * PAGE_SIZE) /* cover ZONE_NORMAL */
    for (j=0; j < PTRS_PER_PMD /* 512 */
      && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
      /* fill up each PMD entry */
      set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
      /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
         Page Size, Global */

      /* each PMD entry covers 2M */
      phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
    }
 }

/* The fourth Page Global Directory entry is then copied into the first entry, so as to
   mirror the mapping of the low physical memory in the first 896 MB of the linear address
   space. This mapping is required in order to complete the initialization of SMP systems:
   when it is no longer necessary, the kernel clears the corresponding page table entries
   by invoking the zap_low_mappings() function, as in the previous cases. */
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

- [*Understanding The Linux Kernel*, Sec. 2.5.5.4, *Final kernel Page Table when RAM size is more than 4096 MB*]
- this code can be used for both PAE and no-PAE situation. That's why the `for` loop

  ```
  for (; i<PTRS_PER_PGD; ++i, ++pgd)
  ```

  is used when `i=3, PTRS_PER_PGD=4`.
- `pgd_index(address)` returns the index of a PGDir entry

  ```
  #define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))
  ```

  ⇒ `(C0000000 >> 30) & (4 - 1)` = 3
  When PAE is enabled, there are 4 entries in `PGDir`. The first 3 entries are for user linear address space. The 4th entry is for kernel space.
- *`#define set_pgd(pgdptr, pgdval) set_pud((pud_t *)(pgdptr), (pud_t) { pgdval })`*
  puds are folded into pgds so this doesn't get actually called, but the define is needed for a generic inline function.

  ```
  set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
  ```

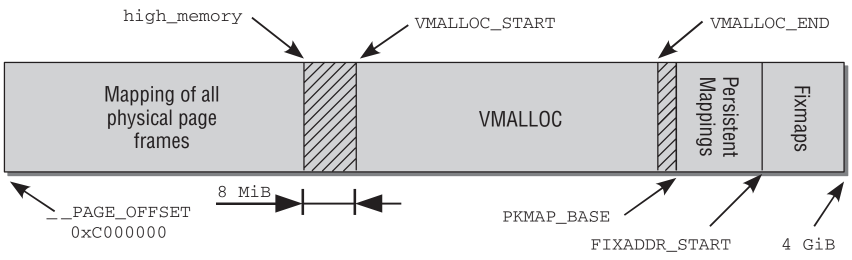
The kernel initializes the first three entries in the PGDir corresponding to the user linear address space with the address of an empty page (empty_zero_page).
Line 417 in arch/i386/kernel/head.S:

```
ENTRY(empty_zero_page)
    .fill 4096,1,0
```

- The 4th entry is initialized with the address of a pmd allocated by invoking alloc_bootmem_low_pages(). There are 512 entries in a pmd,
  - the first 448 entries are filled with the physical address of the first 896MB of RAM.
  - the last 64 entries are reserved for noncontiguous memory allocation [*Understanding The Linux Kernel*, Sec. 8.3, *Noncontiguous Memory Area Management*]

4.5.6 Fix-Mapped Linear Addresses

Division Of The Kernel Address Space
On IA-32 Systems

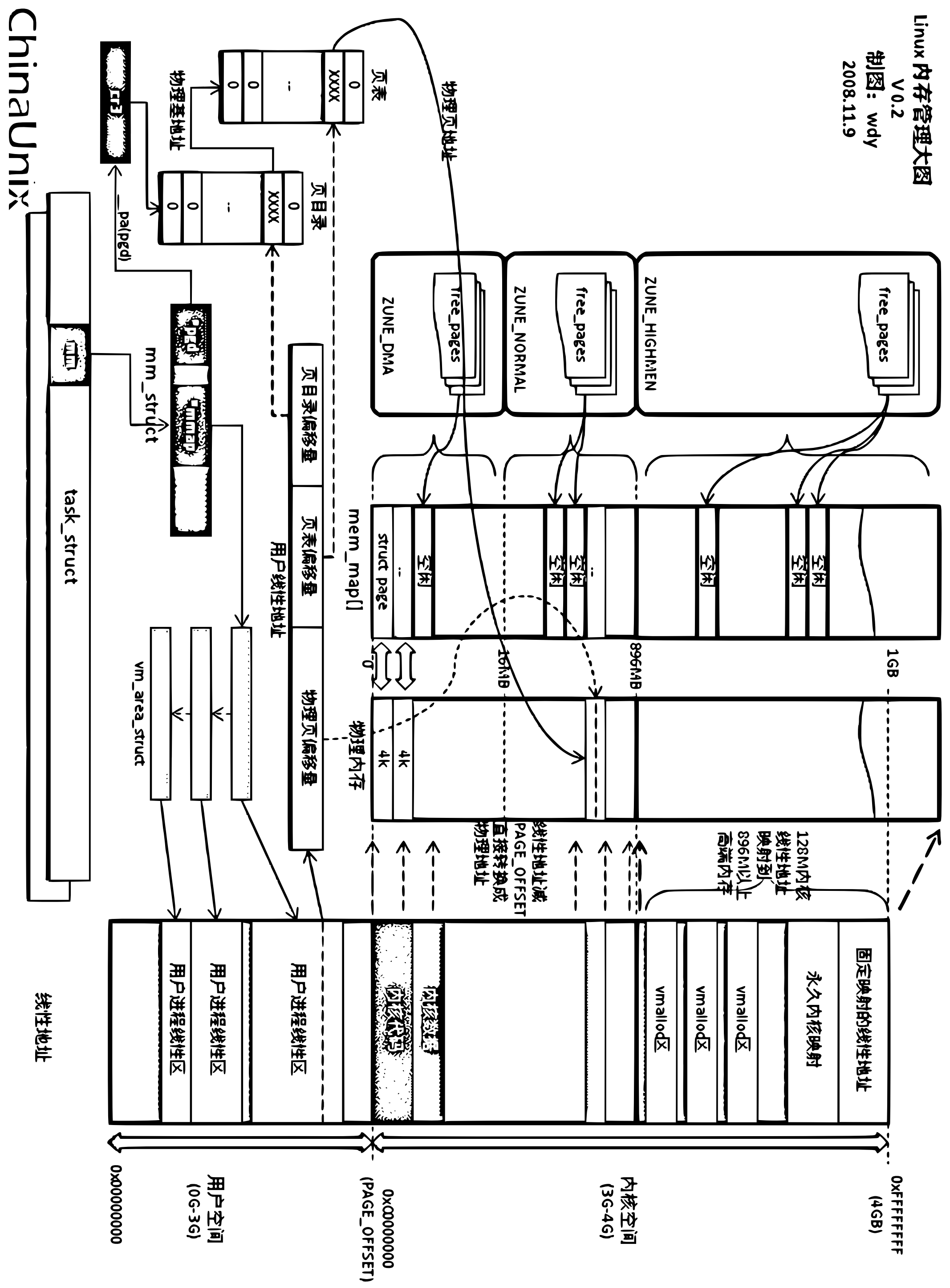


- Virtually contiguous memory areas that are *not* contiguous in physical memory can be reserved in the vmalloc area.
- *Persistent mappings* are used for persistent kernel mapping of highmem page frames.
- *Fixmaps* are virtual address space entries associated with a fixed but freely selectable page in physical address space.

VMALLOC_OFFSET 8MB. This gap acts as a safeguard against any kernel faults. If out of bound addresses are accessed (these are unintentional accesses to memory areas that are no longer physically present), access fails and an exception is generated to report the error. If the vmalloc area were to immediately follow the direct mappings, access would be successful and the error would not be noticed. There should be no need for this additional safeguard in stable operation, but it is useful when developing new kernel features that are not yet mature. [*Professional Linux Kernel Architecture*, Sec. 3.4.2, *Architecture-Specific Setup*, p178]

Fixmaps The advantage of fixmap addresses is that at compilation time, the address acts like a constant whose physical address is assigned when the kernel is booted. Addresses of this kind can be de-referenced faster than when normal pointers are used. The kernel also ensures that the page table entries of fixmaps are not flushed from the TLB during a context switch so that access is always made via fast cache memory. [*Professional Linux Kernel Architecture*, Sec. 3.4.2, *Architecture-Specific Setup*, p180]

See also: [*Memory Management in Linux: Desktop Companion to the Linux Source Code*, Sec. 1.8.3, *Fixmaps*]



5 Processes

5.1 Processes, Lightweight Processes, and Threads

Processes

A process is

- an instance of a program in execution
- a dynamic entity (has lifetime)
- a collection of data structures describing the execution progress
- the unit of system resources allocation

The Linux kernel internally refers to processes as *tasks*.

When A Process Is created

The child

- is almost identical to the parent
  - has a logical copy of the parent's address space
  - executes the same code
- has its own data (stack and heap)

Multithreaded Applications

Threads

- are execution flows of a process
- share a large portion of the application data structures

Lightweight processes (LWP) — Linux way of multithreaded applications

- each LWP is scheduled individually by the kernel
  - no nonblocking syscall is needed
- LWPs may share some resources, like the address space, the open files, and so on.

5.2 Process Descriptor

Process Descriptor

To manage processes, the kernel must have a clear picture of what each process is doing.

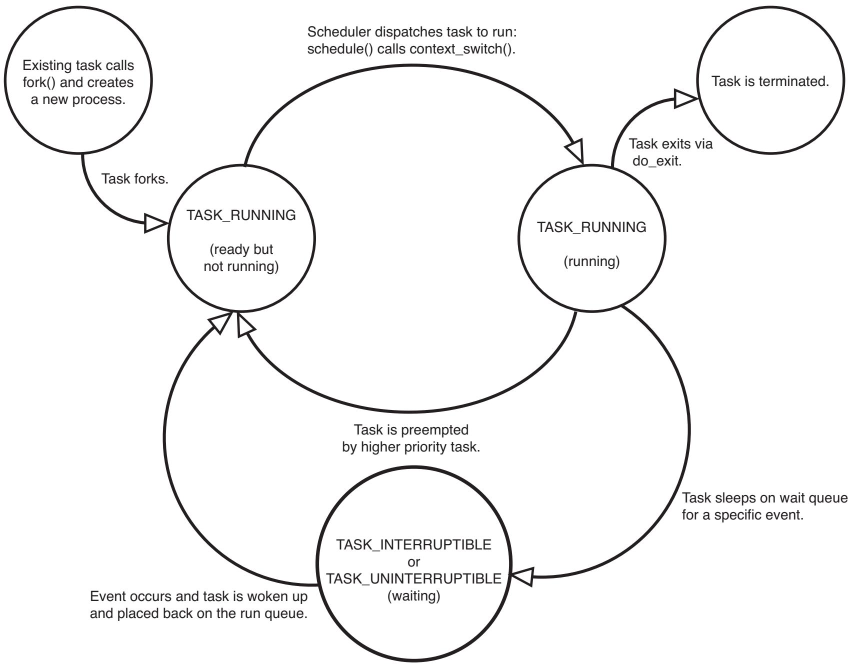
- the process's priority
- running or blocked
- its address space
- files it opened
- ...

Process descriptor: a `task_struct` type structure containing all the information related to a single process.

```
struct task_struct {
  /* 160 lines of code in 2.6.11 */
};
```

5.2.1 Process State

Process State


Existing task calls fork() and creates a new process.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task forks.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

[*Understanding The Linux Kernel*, Sec. 3.2.1, *Process Attribute Related Fields*] The `state` field of `task_struct` is an array of flags, each of which describes a possible process state. In the current Linux version, these states are mutually exclusive, and hence exactly one flag of state always is set; the remaining flags are cleared. The following are the possible process states:

`TASK_RUNNING` The process is either executing on a CPU or waiting to be executed.

`TASK_INTERRUPTIBLE` The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to `TASK_RUNNING`).

`TASK_UNINTERRUPTIBLE` Like `TASK_INTERRUPTIBLE`, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

`TASK_STOPPED` Process execution has been stopped; the process enters this state after receiving a `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal.

`TASK_TRACED` Process execution has been stopped by a debugger. When a process is being monitored by another (such as when a debugger executes a `ptrace()` system call to monitor a test program), each signal may put the process in the `TASK_TRACED` state.

`EXIT_ZOMBIE` Process execution is terminated, but the parent process has not yet issued a `wait4()` or `waitpid()` system call to return information about the dead process. Before the `wait()`-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.

`EXIT_DEAD` The final state: the process is being removed by the system because the parent process has just issued a `wait4()` or `waitpid()` system call for it. Changing its state from `EXIT_ZOMBIE` to `EXIT_DEAD` avoids race conditions due to other threads of execution that execute `wait()`-like calls on the same process.

5.2.2 Identifying a Process

PID AND TGID

- kernel finds a process by its *process descriptor pointer* pointing to a `task_struct`
- users find a process by its PID
- all the threads of a multithreaded application share the same identifier
  tgid: the PID of the thread group leader

```
struct task_struct {
  ...
  pid_t pid;
  pid_t tgid;
  ...
};
```

```
$ ps -eo pgid,ppid,pid,tgid,tid,nlwp,comm --sort pid
```

TODO: Draw a process relationship graph

TODO!

How many PIDs can there be?

- `#define PID_MAX_DEFAULT 0x8000`
- Max PID number = `PID_MAX_DEFAULT - 1 = 32767`

```
$ cat /proc/sys/kernel/pid_max
```

Which are the free PIDs?

```
static pidmap_t pidmap_array[PIDMAP_ENTRIES] =
  {
    [ 0 ... PIDMAP_ENTRIES-1 ] =
    { ATOMIC_INIT(BITS_PER_PAGE), NULL }
  };
```

`pidmap_array` consumes a single page.

Process Descriptor Handling

`thread_union`: 2 consecutive page frames (8K) containing

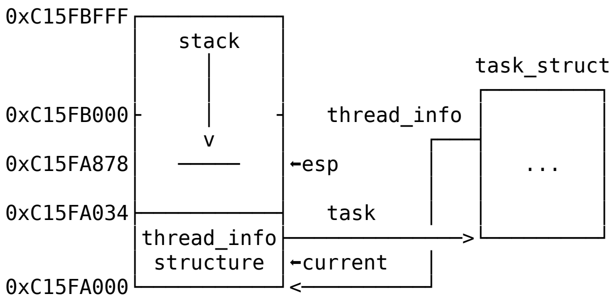
- a process kernel stack
- a `thread_info` structure

```
union thread_union {
  struct thread_info thread_info;
  unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

- The kernel uses the `alloc_thread_info` and `free_thread_info` macros to allocate and release the memory area storing a `thread_info` structure and a kernel stack.
- Benefits of storing `thread_info` and kernel mode stack together [*Understanding The Linux Kernel*, Sec. 3.2.2.2, *Identifying the current process*]:

- Efficiency: the kernel can easily obtain the address of the `thread_info` structure of the process currently running on a CPU from the value of the `esp` register.
- For multi-processor systems: the kernel can easily obtain the address of the `thread_info` structure of the process currently running on a CPU from the value of the `esp` register. Earlier versions of Linux did not store the kernel stack and the process descriptor together. Instead, they were forced to introduce a global static variable called `current` to identify the process descriptor of the running process. On multi-processor systems, it was necessary to define `current` as an array one element for each available CPU.

Kernel Stack



More about kernel stack:

- **Does there exist Kernel stack for each process?** [*Does there exist Kernel stack for each process ?*] In Linux, each task (userspace or kernel thread) has a kernel stack of either 8kb or 4kb, depending on kernel configuration. There are indeed separate stack pointers, however, only one is present in the CPU at any given time; if userspace code is running, the kernel stack pointer to be used on exceptions or interrupts is specified by the task-state segment, and if kernel code is running, the user stack pointer is saved in the context structure located on the kernel stack.
- **kernel stack for Linux process?** [*kernel stack for linux process*]
  - There is just one common kernel memory. In it each process has it's own `task_struct` + kernel stack (by default 8K).
  - In a context switch
    1. the old stack pointer value is stored in the `task_struct` of the old process;
    2. the stack pointer for the new process is read from the `task_struct` of this new process.
  - The kernel stack is a non-shared space where system calls can put their data. If you'd share them between processes, several kernel routines could use the same stack at the same time ⇒ data corruption.
- **kernel stack vs user-mode application stack?** [*kernel stack vs user-mode application stack*] Conceptually, both are the same data structure: a stack. *security concern*
  The reason why there are two different stack per thread is because in user mode, code must not be allowed to mess up kernel memory. When switching to kernel mode, a different stack in memory only accessible in kernel mode is used for return addresses an so on.
  If the user mode had access to the kernel stack, it could modify a jump address (for instance), then do a system call; when the kernel jumps to the previously modified address, your code is executed in kernel mode!
  Also, security-related information/information about other processes (for synchronisation) might be on the kernel stack, so the user mode should not have read access to it either.
- See also: [*Kernel Stack vs User Mode Stack*], [*Kernel Stack Initialization*]

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous flags */
        __u32                   cpu;            /* current CPU */
        __s32                   preempt_count; /* 0 => preemptable, <0 => BUG */


        mm_segment_t            addr_limit;     /* thread address space:
                                                   0-0xBFFFFFFF for user-thead
                                                   0-0xFFFFFFFF for kernel-thread
                                                */
        struct restart_block    restart_block;

        unsigned long           previous_esp;   /* ESP of the previous stack in case
                                                   of nested (IRQ) stacks
                                                */
        __u8                    supervisor_stack[0];
};
```

Why both `task_struct` and `thread_info`?

- There wasn't a `thread_info` in pre-2.6 kernel
- Size matters

More about `thread_info`:

- Since 2.2.x, the `task_struct` is allocated at the bottom of the kernel stack. We can overlap the `task_struct` on the kernel stack because the `task_struct` is a per-task structure exactly as the kernel stack. [*The Linux Process Model*]
- Why are `task_struct` and `thread_info` separate?
  Quote from `http://www.spinics.net/lists/newbies/msg22259.html`:
  Q: There is one `task_struct` and `thread_info` for each process, and they link to each other, right? So why are they separate structs?
  A: `thread_info` is embedded in kernel stack, so we can easily get the `task_struct` using kernel stack.
  Quote from `http://www.spinics.net/lists/newbies/msg22263.html`
  Q: That is a fine workaround for the structs being separate. But why not keep all the data in one struct on the kernel stack, like it was done in 2.4?
  A: `task_struct` is huge. it's around 1.7KB on a 32 bit machine. on the other hand, you can easily see that `thread_info` is much slimmer.
  kernel stack is either 4 or 8KB, and either way a 1.7KB is pretty much, so storing a slimmer struct, that points to `task_struct`, immediately saves a lot of stack space and is a scalable solution.

  size matters
- `supervisor_stack[0]` in `struct thread_info` is a *flexible array member*.
  - *Kerneltrap.org: Regarding `thread_info`*[36]. In the case of `tread_info`: `supervisor_stack` seems to be the kernel stack, in this case an integer number of pages (i.e. 4096 or 8192 bytes) is allocated for the rest of `struct thread_info` + the stack, so
    a) you don't need the real size because you allocate the fixed amount anyway, and
    b) you would have to declare an array of `(4096 - offsetof(struct thread_info, supervisor_stack))` bytes inside the struct itself and that is just not possible.

    Or not declare the stack inside the struct but do offset calculations, but the zero sized array is more elegant and maintainable. There is nothing to fear about that, this is perfectly normal C.
  - See also:
    * [*GCC Manual*, Sec. 6.17, *Arrays of Length Zero*]
    * [*What is the advantage of using zero-length arrays in C?*]
    * *Understand how flexible array members are to be used*[37]
- process descriptors (`task_structs`) are stored in dynamic memory (`ZONE_HIGHMEM`) rather than in the memory area permanently assigned to the kernel
- Kernel stack size, 8K vs. 4K
  - We learned that a process in Kernel Mode accesses a stack contained in the kernel data segment, which is different from the stack used by the process in User Mode. Because kernel control paths make little use of the stack, only a few thousand bytes of kernel stack are required. Therefore, 8 KB is ample space for the stack and the `thread_info` structure. However, when stack and `thread_info` structure are contained in a single page frame, the kernel uses a few additional stacks to avoid the overflows caused by deeply nested interrupts and exceptions [*Understanding The Linux Kernel*, Chapter 4, *Interrupts and Exceptions*]. [*Understanding The Linux Kernel*, Sec. 3.2.2.1, *Process descriptors handling*; Sec. 2.3, *Segmentation in Linux*]
  - (lwn.net: 4K stacks in 2.6) Each process on the system has its own kernel stack, which is used whenever the system goes into kernel mode while that process is running. Since each process requires a kernel stack, the creation of a new process requires an order 1 allocation. So the two-page kernel stacks can limit the creation of new processes, even though the system as a whole is not particularly short of resources. Shrinking kernel stacks to a single page eliminates this problem and makes it easy for Linux systems to handle far more processes at any given time.

    problem in finding 2 consecutive page frames
  - in `linux-2.6.11/include/asm-i386/thread_info.h`

```
#ifdef CONFIG_4KSTACKS
#define THREAD_SIZE (4096)
#else
#define THREAD_SIZE (8192)
#endif
```

`thread_info` and `task_struct` are mutually linked

[36]`http://kerneltrap.org/node/5700`

[37]`https://www.securecoding.cert.org/confluence/display/seccode/MEMxx-C.+Understand+how+flexible+array+members+are+to+be+used`

```c
struct thread_info {
  struct task_struct *task; /* main task structure */
  ...
};

struct task_struct {
  ...
  struct thread_info *thread_info;
  ...
};
```

Identifing The Current Process

Efficiency benefit from `thread_union`

- Easy get the base address of `thread_info` from `esp` register by masking out the 13 least significant bits of `esp`

`current_thread_info()`

```c
/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
  struct thread_info *ti;
  __asm__("andl %%esp, %0;" :"=r" (ti) :"0" (~(THREAD_SIZE - 1)));
  return ti;
}
```

Can be seen as:

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl %ecx,p
```

Each process only gets 8192 bytes of kernel stack, aligned to a 8192-byte boundary, so whenever the stack pointer is altered by a `push` or a `pop`, the low 13 bits are the only part that changes. $2^{13} == 8192$. [*Understanding the getting of task_struct pointer from process kernel stack*]

$$\begin{aligned}\sim(\texttt{THREAD\_SIZE}-1) &= \sim(\texttt{8k}-1)\\ &= \sim(\texttt{0x00002000}-1)\\ &= \sim\texttt{0x00001fff}\\ &= \texttt{11111111111111111110000000000000b}\end{aligned}$$

To get the process descriptor pointer

`current_thread_info()->task`

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl (%ecx),p
```

Because the `task` field is at offset 0 in `thread_info`, after executing these 3 instructions `p` contains the process descriptor pointer.

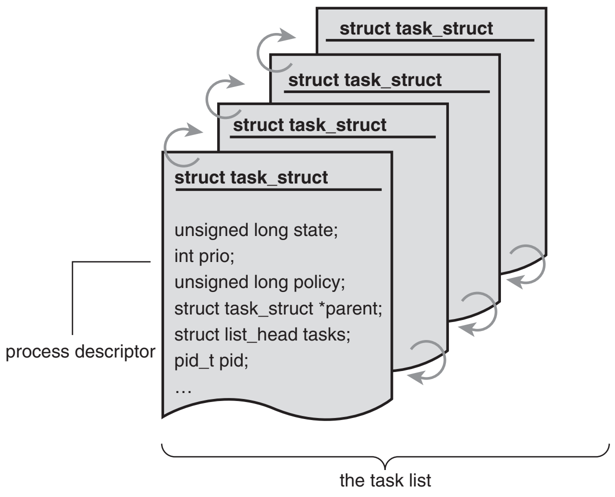
`current` — a marco pointing to the current running task

```c
static inline struct task_struct * get_current(void)
{
        return current_thread_info()->task;
}

#define current get_current()
```

Task List

The kernel stores the list of processes in a circular doubly linked list called the task list.

Swapper The head of this list, `init_task`, process 0.



swapper

Q: Stackoverflow: Why do we need a swapper task in linux?

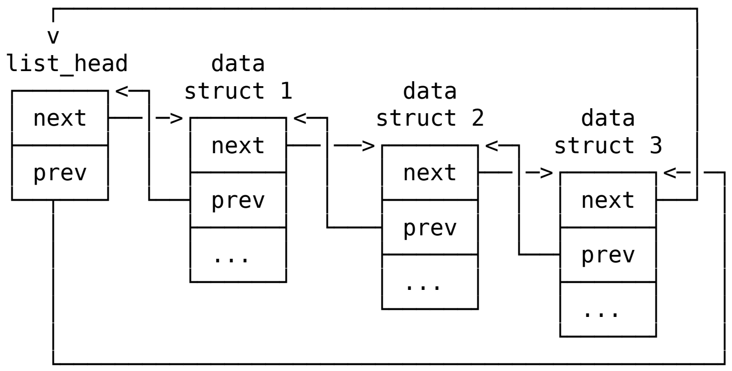
A: The reason is historical and programatic. The idle task is the task running, if no other task is runnable, like you said it. It has the lowest possible priority, so that's why it's running of no other task is runnable.
Programatic reason: This simplifies process scheduling a lot, because you don't have to care about the special case: "What happens if no task is runnable?", because there always is at least one task runnable, the idle task. Also you can count the amount of cpu time used per task. Without the idle task, which task gets the cpu-time accounted no one needs?
Historical reason: Before we had cpus which are able to step-down or go into power saving modes, it HAD to run on full speed at any time. It ran a series of `NOP`-instructions, if no tasks were runnable. Today the scheduling of the idle task usually steps down the cpu by using `HLT`-instructions (halt), so power is saved. So there is a functionality somehow in the idle task in our days.
In Windows you can see the idle task in the process list, it's the idle process.

Q: superuser.com: What is the main purpose of the swapper process in Unix?

A: **It hasn't been a swapper process since the 1990s, and swapping hasn't really been used since the 1970s.**
Unices stopped using swapping a long time ago. They've been demand-paged operating systems for a few decades — since System V R2V5 and 4.0BSD. The swapper process, as was, used to perform process swap operations. It used to swap entire processes — including all of the kernel-space data structures for the process — out to disc and swap them back in again. It would be woken up, by the kernel, on a regular basis, and would scan the process table to determine what swapped-out-and-ready-to-run processes could be swapped in and what swapped-in-but-asleep processes could be swapped out. Any textbook on Unix from the 1980s will go into this in more detail, including the swap algorithm. But it's largely irrelevant to demand-paged Unices, even though they retained the old swap mechanism for several years. (The BSDs tried quite hard to avoid swapping, in favour of paging, for example.)
Process #0 is the first process in the system, hand-crafted by the kernel. It `fork()`s process 1, the first user process. What it does other than that is dependent from what Unix the operating system actually is. As mentioned, the BSDs before FreeBSD 5.0 retained the old swapper mechanism, and process #0 simply dropped into the swapper code in the kernel, the `scheduler()` function, after it had finished system initialization. System V was much the same, except that process #0 was conventionally named `sched` rather than `swapper`. (The names are pretty much arbitrary choices.) In fact, most — possibly all — Unices had a (largely unused) old swapper mechanism that hung around as process #0.
Linux, traditionally, has been somewhat different to the Unices, in that process #0 is the idle process, running `cpu_idle()`. It simply does nothing, in an infinite loop. It exists so that there's always a task ready to be scheduled.
Even this is an out-of-date description. The late 1980s and early 1990s was the advent of multi-threading operating systems, as a result of which process #0 became simply the system process. In the world of old single-threaded Unices, one could only get a separate flow of execution to do a continuous kernel task by `fork()`ing a process. So all of the kernel tasks (vmdaemon, pagedaemon, pagezero, bufdaemon, syncer, ktrace, and so forth on FreeBSD systems, for example) were low-numbered processes, `fork()`ed by process #0 after it `fork()`ed init. In multiple-threaded Unices, it makes no sense to create a whole new process context for something that runs entirely in kernel space, and doesn't need an address space, file descriptor table, and what not all to itself. So all of these tasks became (essentially) threads, sharing the address space of a single system process.
Along the way, several Unices finally lost the old swapper mechanism, that they were trying their utmost to avoid ever using anyway. OpenBSD's initialization code now simply drops into a `while(1) tsleep(⋯);` loop, for example, with no swapping mechanism anywhere.

So nowadays process #0 on a Unix is the system process, which effectively holds a number of kernel threads doing a number of things, ranging from page-out operations, through filesystem cache flushes and buffer zeroing, to idling when there's nothing else to run.

Doubly Linked List



```
struct list_head {
    struct list_head *next, *prev;
};
```

```
struct task_struct {
    ...
    struct list_head tasks;
    ...
};
```

List operations

`SET_LINKS` insert into the list
`REMOVE_LINKS` remove from the list
`for_each_process` scan the whole process list

```
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

`list_for_each` iterate over a list

```
#define list_for_each(pos, head)                                    \
  for (pos = (head)->next; prefetch(pos->next), pos != (head); \
       pos = pos->next)
```

Example: Iterate over a process' children

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
  task = list_entry(list, struct task_struct, sibling);
  /* task now points to one of current's children */
}
```

Expend the macro `list_for_each()`:

```
struct task_struct *task;
struct list_head *list;
for (list = (&current->children)->next;
             prefetch(list->next), list != (&current->children);
             list = list->next)
{
    task = list_entry(list, struct task_struct, sibling);
}
```

- (Linux Kernel Linked List Explained) We need to access *the item itself* not the variable "`list`" in the item! macro `list_entry()` does just that.

  ```
  tmp = list_entry(pos, struct kool_list, list);
  ```

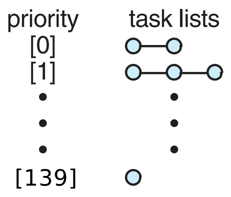
  Given
  1. a pointer to `struct list_head`,
  2. type of data structure it is part of, and
  3. it's name (`struct list_head`'s name in the data structure)

  it returns a pointer to the data structure in which the pointer is part of.
- [*Linux Kernel Development*, Sec 3.2.6, *The Process Family Tree*].

A task can be in multiple lists

```
struct task_struct {
  struct list_head run_list;
  struct list_head tasks;
  struct list_head ptrace_children;
  struct list_head ptrace_list;
  struct list_head children; /* list of my children */
  struct list_head sibling;  /* linkage in my parent's children list */
}
```

The List Of `TASK_RUNNING` Processes



- Each CPU has its own runqueue
- Each runqueue has 140 lists
- One list per process priority
- Each list has zero to many tasks

```
struct task_struct {
  ...
  int prio, static_prio;
  struct list_head run_list;
  prio_array_t *array;
  ...
};
```

`prio` The dynamic priority of a process is a value that depends on the processes' scheduling history and the specified `nice` value. It is updated at `sleep` time, which is when the process is not being executed and when timeslice is used up. This value, `prio`, is related to the value of the `static_prio` field described next. The `prio` field holds `+/- 5` of the value of `static_prio`, depending on the process' history; it will get a +5 bonus if it has slept a lot and a -5 handicap if it has been a processing hog and used up its timeslice. [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Sec. 3.2.2.1, *`prio`*]

`static_prio` is equivalent to the `nice` value. The default value of `static_prio` is `MAX_PRIO - 20`. In our kernel, `MAX_PRIO` defaults to 140. [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Sec 3.2.2.2, `static_prio`]

`run_list` The trick used to achieve the scheduler speedup consists of splitting the runqueue in many lists of runnable processes, one list per process priority. Each `task_struct` descriptor includes a `run_list` field of type `list_head`. If the process priority is equal to k (a value ranging between 0 and 139), the `run_list` field links the process descriptor into the list of runnable processes having priority k. [*Understanding The Linux Kernel*, Sec. 3.2.2.5, *The lists of `TASK_RUNNING` processes*]

Each Runqueue Has A `prio_array_t` Struct

```
typedef struct prio_array prio_array_t;

struct prio_array {
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
```

`nr_active`: The number of process descriptors linked into the lists (the whole runqueue)
`bitmap`: A priority bitmap. Each flag is set if the priority list is not empty
`queue`: The 140 heads of the priority lists

- Each *runqueue (NOT each priority)* has a `prio_array_t`. [*Understanding The Linux Kernel*, Sec. 3.2.2.5, *The lists of `TASK_RUNNING` processes*] As we'll see, the kernel must preserve a lot of data for every runqueue in the system; however, the main data structures of a runqueue are the lists of process descriptors belonging to the runqueue; all these lists are implemented by a single `prio_array_t` data structure.

To Insert A Task Into A Runqueue List

```
static void enqueue_task(struct task_struct *p, prio_array_t *array)
{
  ...
  list_add_tail(&p->run_list, &array->queue[p->prio]);
  __set_bit(p->prio, array->bitmap);
  array->nr_active++;
  p->array = array;
}
```

`prio`: priority of this process
`array`: a pointer pointing to the `prio_array_t` of this runqueue

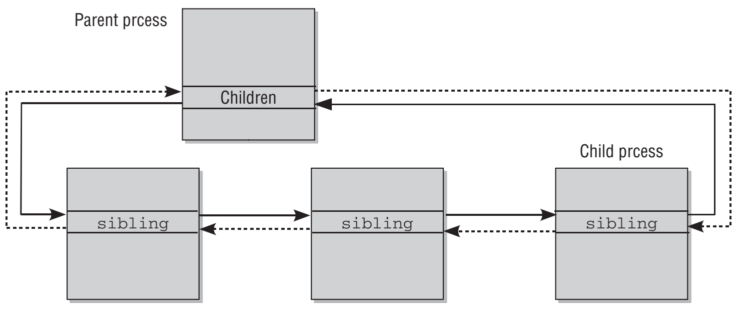
- To removes a process descriptor from a runqueue list, use `dequeue_task(p,array)` function.

5.2.3 Relationships Among Processes

Family relationship

```
struct task_struct {
  ...
  struct list_head children; /* list of my children */
  struct list_head sibling;  /* linkage in my parent's children list */
  ...
};
```

children: is the list head for the list of all child elements of the process
sibling: is used to link siblings with each other



Other Relationships

A process can be:

- a leader of a process group or of a login session
- a leader of a thread group
- tracing the execution of other processes

```
struct task_struct {
  ...
  pid_t tgid;
  ...
  struct task_struct *group_leader; /* threadgroup leader */
  ...
  struct list_head ptrace_children;
  struct list_head ptrace_list;
  ...
};
```

The Pid Hash Table And Chained Lists

PID ⇒ process descriptor pointer?

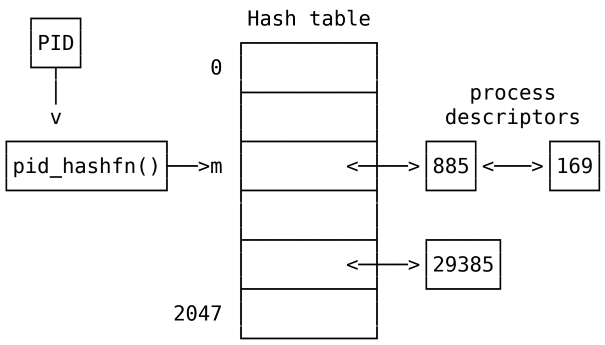
- Scanning the process list? — too slow
- Use hash tables

Four hash tables have been introduced

Why 4? For 4 types of PID $\left.\begin{array}{c}\texttt{PID}\\\texttt{TGID}\\\texttt{PGID}\\\texttt{SID}\end{array}\right\}\Rightarrow$ `task_struct`

Hash Table Type	Field Name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

Collision
Multiple PIDs can be hashed into one table index



- Chaining is used to handle colliding PIDs
- No collision if the table is 32768 in size! But...

The PID is transformed into a table index using the `pid_hashfn` macro, which expands to:

```
#define pid\_hashfn(x) hash\_long((unsigned long) x, pidhash\_shift)
```

The `pidhash_shift` variable stores the length in bits of a table index (11, in our example). The `hash_long()` function is used by many hash functions; on a 32-bit architecture it is essentially equivalent to:

```
unsigned long hash_long(unsigned long val, unsigned int bits)
{
    unsigned long hash = val * 0x9e370001UL;
    return hash >> (32 - bits);
}
```

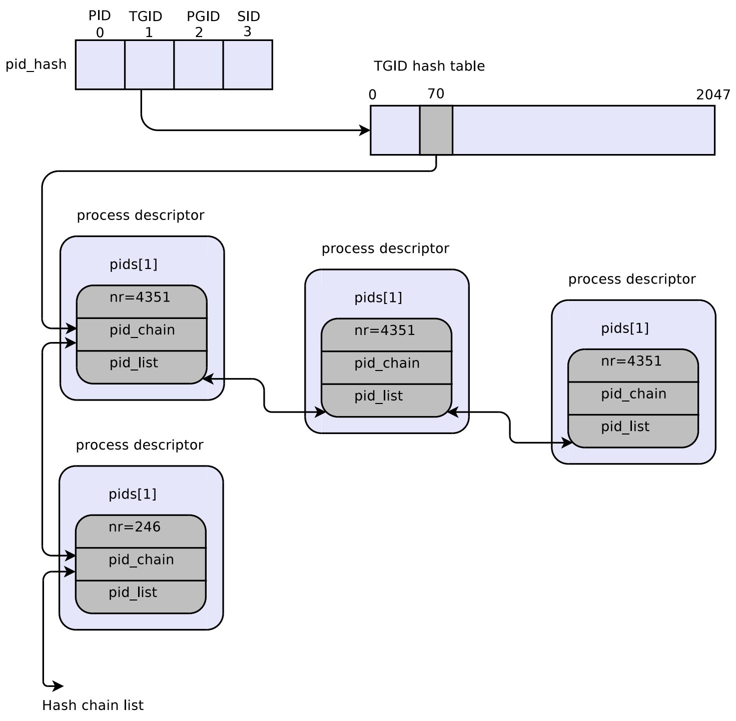
Because in our example `pidhash_shift` is equal to 11, `pid_hashfn` yields values ranging between 0 and $2^{11} - 1 = 2047$.

The `pid` data structure

```
struct pid
{
  int nr;
  struct hlist_node pid_chain;
  struct list_head pid_list;
};
```

```
struct task_struct{
  ...
  struct pid pids[PIDTYPE_MAX];
  ...
}
```

PID Hash Tables



`kernel/pid.c` — Operations

- `do_each_task_pid(nr, type, task)`
- `while_each_task_pid(nr, type, task)`
- `find_task_by_pid_type(type, nr)`
- `find_task_by_pid(nr)`
- `attach_pid(task, type, nr)`
- `detach_pid(task, type)`
- `next_thread(task)`

5.2.4 How Processes Are Organized

Wait Queues

- A wait queue represents a set of sleeping processes, which are woken up by the kernel when some condition becomes true.
- Wait queues are implemented as doubly linked lists whose elements include pointers to process descriptors.

Each wait queue is identified by a `__wait_queue_head`

```
struct __wait_queue_head {
        spinlock_t lock;
        struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

lock: avoid concurrent accesses.

Elements of a wait queue list are of type `wait_queue_t`:

```
struct __wait_queue {
        unsigned int flags;
        struct task_struct * task;
        wait_queue_func_t func;
        struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

`task`: address of this sleeping process
`task_list`: which wait queue are you in?
`flags`: 1 – exclusive; 0 – nonexclusive;
`func`: how it should be woken up?

5.2.5 Process Resource Limits

Limiting the resource use of a process

- The amount of system resources a process can use are stored in the `current->signal->rlim` field.
- `rlim` is an array of elements of type `struct rlimit`, one for each resource limit.

```
struct rlimit {
        unsigned long   rlim_cur;
        unsigned long   rlim_max;
};
```

`rlim_cur`: the current resource limit for the resource

e.g. `current->signal->rlim[RLIMIT_CPU].rlim_cur` — the current limit on the CPU time of the running process.

`rlim_max`: the maximum allowed value for the resource limit

Resource Limits

RLIMIT_AS	The maximum size of process address space
RLIMIT_CORE	The maximum core dump file size
RLIMIT_CPU	The maximum CPU time for the process
RLIMIT_DATA	The maximum heap size
RLIMIT_FSIZE	The maximum file size allowed
RLIMIT_LOCKS	Maximum number of file locks
RLIMIT_MEMLOCK	The maximum size of nonswappable memory
RLIMIT_MSGQUEUE	Maximum number of bytes in POSIX message queues
RLIMIT_NOFILE	The maximum number of open file descriptors
RLIMIT_NPROC	The maximum number of processes of the user
RLIMIT_RSS	The maximum number of page frames owned by the process
RLIMIT_SIGPENDING	The maximum number of pending signals for the process
RLIMIT_STACK	The maximum stack size

5.3 Process Switch

Process execution context: all information needed for the process execution
Hardware context: the set of registers used by a process

Where is the hardware context stored?

- partly in the process descriptor (PCB)
- partly in the Kernel Mode stack

Process switch

- saving the hardware context of `prev`
- replacing it with the hardware context of `next`

Process switching occurs only in Kernel Mode.

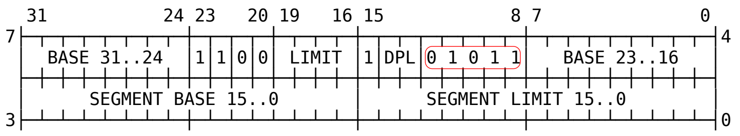
Task State Segment (TSS)

- For storing hardware contexts
- One TSS for each process (Intel's design)
- Hardware context switching
  - `far jmp` to the TSS of `next`

Linux doesn't use hardware context switch

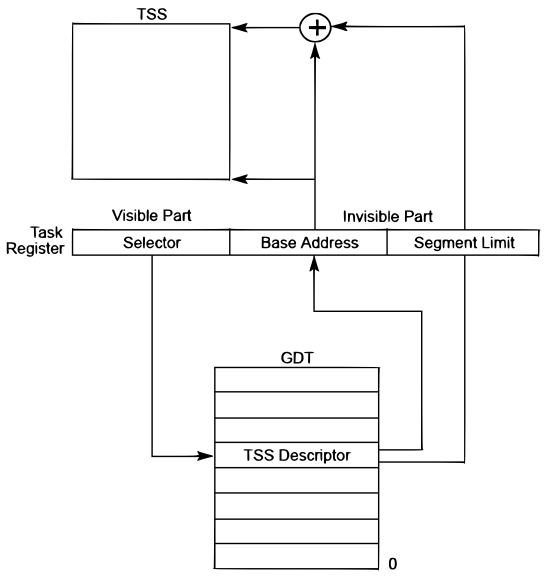
- One TSS for each CPU
  - The address of the kernel mode stack
  - I/O permission bitmap

Task State Segment Descriptor (TSSD)



- `S` bit set to 0;
- `Type` bits set to 9/11;
- `Busy` bit set to 1.

The Task Register (`tr`)



Where to save the hardware context?

```
struct task_struct{
  ...
  struct thread_struct thread;
  ...
}
```

- `thread_struct` includes fields for most of the CPU registers, except the general-purpose registers such as `eax, ebx`, etc., which are stored in the Kernel Mode stack.

Performing The Process Switch
`schedule()`

Two steps:

1. Switching the Page Global Directory
2. Switching the Kernel Mode stack and the hardware context

`switch_to(prev,next,last)`

- in any process switch three processes are involved, not just two

Stackoverflow: **How does `schedule()` + `switch_to()` actually work?**

- When a process runs out of time-slice, the flag `TIF_NEED_RESCHED` is set by `scheduler_tick()`. (Called from the timer interrupt handler, `set_tsk_need_resched()`)
- The kernel checks the flag, sees that it is set, and calls `schedule()` to switch to a new process. This flag is a message that schedule should be invoked as soon as possible because another process deserves to run. Upon returning to user-space or returning from an interrupt, the `TIF_NEED_RESCHED` flag is checked. If it is set, the kernel invokes the scheduler before continuing.

5.4 Creating Processes

- [*Professional Linux Kernel Architecture*, Chapter 2, *Process Management And Scheduling*]
- [*The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Section 3.3. *Process Creation: fork(), vfork(), and clone() System Calls*]
- [*Linux Kernel Development*, Section 3.2, *Process Creation*]

Creating Processes

The `clone()` system call

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

The traditional `fork()` system call

`clone(func, child_stack, SIGCHLD, NULL);`

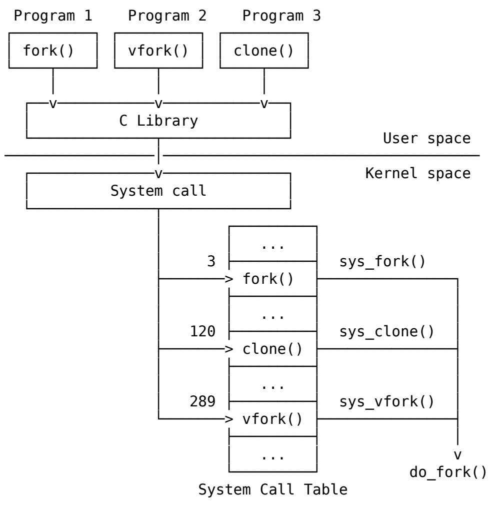
- `child_stack`: parent stack pointer (copy-on-write)

`vfork()`

`clone(func, child_stack, CLONE_VM|CLONE_VFORK|SIGCHLD, NULL);`

- `child_stack`: parent stack pointer (copy-on-write)

The `do_fork()` function does the real work



`do_fork()` calls `copy_process()` to make a copy of process descriptor

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
{
  struct task_struct *p;
  ...
  long pid = alloc_pidmap();
  ...
  p = copy_process(clone_flags, stack_start, regs,
                   stack_size, parent_tidptr,
                   child_tidptr, pid);
  ...
  return pid;
}
```

`copy_process()`

1. `dup_task_struct()`: creates
   - a new kernel mode stack
   - `thread_info`
   - `task_struct`

   Values are identical to the parent
2. is `current->signal->rlim[RLIMIT_NPROC].rlim_cur` confirmed?
3. Update child's `task_struct`
4. Set child's state to `TASK_UNINTERRUPTABLE`
5. `copy_flags()`: update flags in `task_struct`
6. `get_pid()` (check `pidmap_array` bitmap)
7. Duplicate or share resources (opened files, FS info, signal, ...)
8. `return p;`

See also: [*Linux Kernel Development*, Sec 3.2.2, *Forking*]

Creating A Kernel Thread

`kernel_thread()` is similar to `clone()`

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
  ...
  return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);
}
```

- Typically, a kernel thread continues executing its initial function forever (or at least until the system reboots, but with Linux you never know). The initial function usually implements a loop in which the kernel thread wakes up as needed, performs its duties, and then returns to sleep [*Linux Kernel Development*, Sec 3.4.2, *Kernel Threads*].

Process 0

Process 0 is a kernel thread created from scratch during the initialization phase.

- Also called *idle process*, or *swapper process*
- Its data structures are *statically* allocated

`start_kernel()`

- Initializes all the data structures
- Enables interrupts
- Creates another kernel thread — *process 1, the `init` process*

Call graph

```
start_kernel()
  └─> rest_init()
         ├─> kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND)
         └─> cpu_idle()
```

- After having created the *init* process, *process 0* executes the `cpu_idle()` function.
- Process 0 is selected by the scheduler only when there are no other processes in the `TASK_RUNNING` state.
- In multiprocessor systems there is a process 0 for each CPU.

Process 1

- Created via `kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);`
- PID is 1
- shares all per-process kernel data structures with process 0
- starts executing the `init()` function
  - completes the initialization of the kernel
- `init()` invokes the `execve()` system call to load the executable program `init`
  - As a result, the *init kernel thread* becomes a regular process having its own per-process kernel data structure
- The init process stays alive until the system is shut down

5.5 Destroying Processes

Process Termination

- Usual way: call `exit()`
  - The C compiler places a call to `exit()` at the end of `main()`.
- Unusual way: `Ctrl-C ...`

All process terminations are handled by `do_exit()`

- `tsk->flags |= PF_EXITING;` to indicate that the process is being eliminated
- `del_timer_sync(&tsk->real_timer);` to remove any kernel timers
- `exit_mm(), exit_sem(), __exit_files(), __exit_fs(), exit_namespace(), exit_thread()`: free pointers to the kernel data structures
- `tsk->exit_code = code;`
- `exit_notify()` to send signals to the task's parent
  - re-parents its children
  - sets the task's state to `TASK_ZOMBIE`
- `schedule()` to switch to a new process

Process Removal
Cleaning up after a process and removing its process descriptor are separate.

Clean up

- done in `do_exit()`
- leaves a zombie
  - To provide information to its parent
  - The only memory it occupies is its kernel stack, the `thread_info` structure, and the `task_struct` structure.

Removal

- `release_task()` is invoked by
  either `do_exit()` if the parent didn't wait
  or `wait4()/waitpid()`
- `free_uid()`
- `unhash_process`: to remove the process from the pidhash and from the task list
- `put_task_struct()`
  - free the pages containing the process's kernel stack and `thread_info` structure
  - de-allocate the slab cache containing the `task_struct`

6 Process Scheduling

6.1 Multitasking

1. Cooperative multitasking
   Yielding a process voluntarily suspends itself
2. Preemptive multitasking
   Preemption involuntarily suspending a running process
   Timeslice the time a process runs before it's preempted
   - usually dynamically calculated
   - used as a configurable system policy

   But Linux's scheduler is different

6.2 Linux's Process Scheduler

up to 2.4: simple, scaled poorly

- $O(n)$
- non-preemptive
- single run queue (cache? SMP?)

from 2.5 on: $O(1)$ scheduler
- 140 priority lists — scaled well
- one run queue per CPU — true SMP support
- preemptive
- ideal for large server workloads
- showed latency on desktop systems

from 2.6.23 on: Completely Fair Scheduler (CFS)
- improved interactive performance

up to 2.4: [*Understanding The Linux Kernel*, Sec. 7.2, *The Scheduling Algorithm*] The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward: at every process switch the kernel scanned the list of runnable processes, computed their priorities, and selected the ”best” process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly, that is, it spends too much time in high-end systems running thousands of processes.
No true SMP all processes share the same run-queue
Cold cache if a process is re-scheduled to another CPU

6.3 Scheduling Policy

Must attempt to satisfy two conflicting goals:

1. fast process response time (low latency)
2. maximal system utilization (high throughput)

Linux tries

1. favoring I/O-bound processes over CPU-bound processes
2. doesn't neglect CPU-bound processes

Process Priority

Usually,

- processes with a higher priority run before those with a lower priority
- processes with the same priority are scheduled *round-robin*
- processes with a higher priority receive a longer time-slice

Linux implements two priority ranges:

1. **Nice value**: $-20 \sim +19$ (default 0)
   - large value ⇒ lower priority
   - lower value ⇒ higher priority ⇒ get larger proportion of a CPU

   ```
   $ ps -el
   ```

   The *nice value* can be used as
   - a control over the *absolute* time-slice (e.g. MAC OS X), or
   - a control over the *proportion* of time-slice (Linux)
2. **Real-time**: $0 \sim 99$
   - higher value ⇒ greater priority

   ```
   $ ps -eo state,uid,pid,rtprio,time,comm
   ```

Time-slice

too long: poor interactive performance
too short: context switch overhead
I/O-bound processes: don't need longer time-slices (prefer short queuing time)
CPU-bound processes: prefer longer time-slices (to keep their caches hot)
Apparently, any long time-slice would result in poor interactive performance.

Problems With Nice Value

`if` two processes A and B
A: $NI = 0, t = 100ms$
B: $NI = 20, t = 5ms$

`then`, the CPU share
A: gets $\frac{100}{105} = 95\%$
B: gets $\frac{5}{105} = 5\%$

What if two $B_{ni=20}$ running?
Good news: Each gets 50%
Bad news: This '50%' is $\frac{5}{10}$, NOT $\frac{52.5}{105}$
Context switch twice every 10ms!

Comparing
$P_{ni=0}$ gets 100ms
$P_{ni=1}$ gets 95ms

With
$P_{ni=19}$ gets 10ms
$P_{ni=20}$ gets 5ms

This behavior means that "nicing down a process by one" has wildly different effects depending on the starting nice value.

6.3.1 Linux's CFS

Completely Fair Scheduler (CFS)

For a perfect (unreal) multitasking CPU

- n runnable processes can run at the same time
- each process should receive $\frac{1}{n}$ of CPU power

For a real world CPU

- can run only a single task at once — unfair

☺ while one task is running
☹☹ the others have to wait

- `p->wait_runtime` is the amount of time the task should now run on the CPU for it becomes completely fair and balanced.
  ☺ on ideal CPU, the `p->wait_runtime` value would always be zero
- CFS always tries to run the task with the largest `p->wait_runtime` value
- kerneltrap: Linux: Discussing the Completely Fair Scheduler

CFS

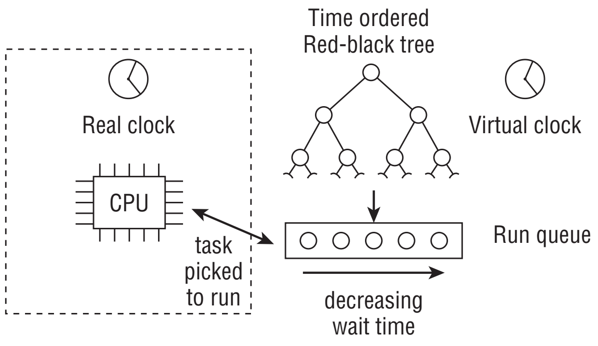
In practice it works like this:

- While a task is using the CPU, its `wait_runtime` decreases

```
wait_runtime = wait_runtime - time_running
```

if: its `wait_runtime` $\neq$ $\mathtt{MIN}_{\mathtt{wait\_runtime}}$ (among all processes)
then: it gets preempted

- Newly woken tasks (`wait_runtime = 0`) are put into the tree more and more to the right
- slowly but surely giving a chance for every task to become the 'leftmost task' and thus get on the CPU within a deterministic amount of time



- the run queue is sorted by waiting time with a red-black tree
- the leftmost node in the tree is picked by the scheduler
- An outstanding feature of the Linux scheduler is that it does not require the concept of time slices, at least not in the traditional way. Classical schedulers compute time slices for each process in the system and allow them to run until their time slice is used up. When all time slices of all processes have been used up, they need to be recalculated again. The current scheduler, in contrast, considers only the wait time of a process — that is, how long it has been sitting around in the run-queue and was ready to be executed. The task with the gravest need for CPU time is scheduled. [*Professional Linux Kernel Architecture*, p84]
  ... but what do *fair* and *unfair* with respect to CPU time mean? Consider an ideal computer that can run an arbitrary number of tasks in parallel: If N processes are present on the system, then each one gets $\frac{1}{N}$ of the total computational power, and all tasks really execute physically parallel. Suppose that a task requires 10 minutes to complete its work. If 5 such tasks are simultaneously present on a perfect CPU, each will get 20 percent of the computational power, which means that it will be running for 50 instead of 10 minutes. However, all 5 tasks will finish their job after exactly this time span, and none of them will have ever been inactive!
  If multitasking is simulated by running one process after another, then the process that is currently running is favored over those waiting to be picked by the scheduler — the poor waiting processes are being treated unfairly. The unfairness is directly proportional to the waiting time.
  Every time the scheduler is called, it picks the task with the highest waiting time and gives the CPU to it. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system.

The virtual clock

Time passes slower on this clock than in real time
more processes waiting $\Rightarrow$ more slower

Example

if: 4 processes in run queue
then: the virtual clock speed is $\frac{1}{4}$ of the real clock
if: a process sitting in the queue for 20s in real time

then: resulting to 5s in virtual time
if: the 4 processes executing for 5s each
then: the CPU will be busy for 20s in real time

- The virtual clock is a per process clock. Whenever the CPU comes, this clock starts ticking. And it stops ticking immediately when the CPU is going away.

To sort tasks on the red-black tree

```
fair_clock - wait_runtime
```

`fair_clock` The virtual time, e.g. 5s in the previous example
`wait_runtime` Fairness imbalance measure

To move a node rightward in the red-black tree

```
wait_runtime = wait_runtime - time_running
```

`time_running` When a task is allowed to run, the interval during which it has been running

CFS

Example:
Assuming *targeted latency* is 20ms. If we have
2 processes: each gets 10ms
4 processes: each gets 5ms
20 processes: each gets 1ms
∞ **processes:** each gets 1ms (to avoid unacceptable context switching costs)

See also: [*Linux Kernel Development*, Sec. 4.4, *The Linux Scheduling Algorithm*, p49]

Example
A system with two processes running:
1. a text editor, say, Emacs (I/O-bound)
2. gcc is compiling the kernel source (CPU-bound)

if: they both have the same nice value
then: the proportion they get would be 50%-50%
Consequence:
- Emacs uses far less than 50% of CPU
- gcc can enjoy more than 50% of CPU freely

When Emacs wakes up
1. CFS notes that it has 50% of CPU, but uses very little of it (far less than gcc)
2. CFS preempts gcc and enables Emacs to run immediately

Thus, better interactive performance.

6.4 Linux Scheduling Algorithm

Scheduler Classes
Different, pluggable algorithms coexist
- Each algorithm schedules its own type of processes
- Each scheduler class has a priority

`SCHED_FIFO`
`SCHED_RR`
`SCHED_NORMAL`

Example: nice value difference
Assume:
1. nice value 5 pts up results in a $\frac{1}{3}$ penalty
2. targeted latency is again 20ms
3. 2 processes in the system

Then:
- $P_{ni=0}$ gets 15ms; $P_{ni=5}$ gets 5ms
- $P_{ni=10}$ gets 15ms; $P_{ni=15}$ gets 5ms
- Absolute nice values no longer affect scheduling decision
- Relative nice values does

See also: [*Linux Kernel Development*, Sec. 4.4, *The Linux Scheduling Algorithm*, p49]

6.5 The Linux Scheduling Implementation

Ref: [*Linux Kernel Development*, Sec. 4.5, *The Linux Scheduling Implementation*]

- Wikipedia: Completely Fair Scheduler
- IBM developerworks: Multiprocessing with the Completely Fair Scheduler
- Scheduling in the Linux kernel –RSDL/SD
- kerneltrap: Linux: The Completely Fair Scheduler

See also: [*Understanding The Linux Kernel*, Sec. 7.2, *Preemption*].

Base Time Quantum

$O(1)$ scheduler

$$\begin{array}{c}\text{base time quantum}\\ \text{(ms)}\end{array} = \begin{cases} (140 - \text{static priority}) \times 20 & if \text{ static priority} < 120, \\ (140 - \text{static priority}) \times 5 & if \text{ static priority} \geq 120. \end{cases}$$

Major Components of CFS

- Time Accounting
- Process Selection
- The Scheduler Entry Point
- Sleeping and Waking Up

Time accounting

`sched_entity` keeps track of process accounting (`task_struct -> se`)

```
struct sched_entity {
  struct load_weight      load;          /* for load-balancing */
  struct rb_node          run_node;
  struct list_head        group_node;
  unsigned int            on_rq;

  u64                     exec_start;
  u64                     sum_exec_runtime;
  u64                     vruntime;
  u64                     prev_sum_exec_runtime;

  u64                     last_wakeup;
  u64                     avg_overlap;

  u64                     nr_migrations;

  u64                     start_runtime;
  u64                     avg_wakeup;

  /* many state variables elided, enabled only if CONFIG_SCHEDSTATS is set */
};
```

The Virtual Runtime

`vruntime` stores the *virtual runtime* of a process. On an ideal processor, all tasks' `vruntime` would be identical. Accounting is done in `update_curr()` and `__update_curr()`

- `update_curr()` (Line 518 in `kernel/sched_fair.c`)
- `__update_curr()` (Line 503 in `kernel/sched_fair.c`)
- [*Linux Kernel Development*, p51-p52]

Process Selection

The core of CFS algorithm Pick the process with the smallest `vruntime`

- run the process represented by the leftmost node in the `rbtree`

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
  struct rb_node *left = cfs_rq->rb_leftmost;
  if (!left)
    return NULL;
  return rb_entry(left, struct sched_entity, run_node);
}
```

Adding Processes to the Tree

- Happens when a process wakes up or is created
- `enqueue_entity()` and `__enqueue_entity()`

See also: [*Linux Kernel Development*, p54-55]

Removing Processes from the Tree

- Happens when a process blocks or terminates
- `dequeue_entity()` and `__dequeue_entity()`

The Scheduler Entry Point

Sleeping and Waking Up

References

[1] R. Love, *Linux Kernel Development*, Addison-Wesley, **2010**.

[2] Wikipedia, Unix — Wikipedia, The Free Encyclopedia, [Online; accessed 19-September-2017], **2017**.

[3] D. Bovet, M. Cesati, *Understanding The Linux Kernel*, 3rd ed., OReilly, **2005**.

[4] Stackoverflow, Userspace process preempts kernel thread?, [Online; accessed 18-Apr-2015], **2011**.

[5] Stackoverflow, Threadsafe vs re-entrant, [Online; accessed 18-Apr-2015], **2009**.

[6] Stackoverflow, what is the difference between re-entrant function and recursive function in C?, [Online; accessed 18-Apr-2015], **2008**.

[7] Wikipedia, Unix signal — Wikipedia, The Free Encyclopedia, [Online; accessed 18-April-2015], **2015**.

[8] W. R. Stevens, S. A. Rago, *Advanced programming in the UNIX environment*, Addison-Wesley, **2013**.

[9] Stackexchange, What are session leaders in PS?, [Online; accessed 18-Apr-2015], **2011**.

[10] Silberschatz, Galvin, Gagne, *Operating System Concepts Essentials*, John Wiley & Sons, **2011**.

[11] T. D. K. H. Project, *Debian Linux Kernel Handbook*, Free software Foundation, **2013**.

[12] Wikipedia, Inline function — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[13] RJK, Inline Functions In C, [Online; accessed 19-Apr-2015].

[14] Wikipedia, Inline assembler — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[15] C. Rodriguez, G. Fischer, S. Smolski, *The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*, Prentice Hall Professional Technical Reference, **2005**.

[16] kernelnewbies.org, FAQ/LikelyUnlikely, [Online; accessed 19-Apr-2015], **2006**.

[17] Wikipedia, Linux kernel oops — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2014**.

[18] stackoverflow, Use of floating point in the Linux kernel, [Online; accessed 19-Apr-2015], **2012**.

[19] blogspot.com, Linux Kernel and Floating Point, [Online; accessed 19-Apr-2015], **2011**.

[20] J. Corbet, G. Kroah-Hartman, A. Rubini, *Linux Device Drivers*, 3rd ed., O'Reilly, **2005**.

[21] stackoverflow, Array versus linked-list, [Online; accessed 19-Apr-2015], **2014**.

[22] kernelnewbies.org, FAQ/LinkedLists, [Online; accessed 19-Apr-2015], **2014**.

[23] isis.poly.edu, Linux Kernel Linked List Explained, [Online; accessed 19-Apr-2015], **2005**.

[24] kernelnewbies.org, FAQ/LinkedLists, http://kernelnewbies.org/FAQ/LinkedLists, (Accessed on 2016-06-26).

[25] ubuntuforums.org, link list implementation in linux kernel, http://ubuntuforums.org/archive/index.php/t-1591281.html, (Accessed on 2016-06-26).

[26] stackoverflow, LIST_HEAD_INIT vs INIT_LIST_HEAD, http://stackoverflow.com/questions/10262017/linux-kernel-list-list-head-init-vs-init-list-head, (Accessed on 2016-06-26).

[27] Wikipedia, C dynamic memory allocation — Wikipedia, The Free Encyclopedia, [Online; accessed 25-April-2015], **2015**.

[28] Linuxmm, PageAllocation, [Online; accessed 18-Apr-2015], **2013**.

[29] ubuntuforums.org, link list implementation in linux kernel easy question syntax not clear, [Online; accessed 19-Apr-2015], **2010**.

[30] stackoverflow, What's the use of do while(0) when we define a macro?, [Online; accessed 19-Apr-2015], **2009**.

[31] stackoverflow, Do-While and if-else statements in C/C++ macros, [Online; accessed 19-Apr-2015], **2010**.

[32] kernelnewbies.org, FAQ/ContainerOf, [Online; accessed 19-Apr-2015], **2006**.

[33] blogspot.com, Linux Kernel: 🄵大又好用的 list_head 結構, [Online; accessed 18-Apr-2015], **2007**.

[34] Wikipedia, Circular buffer — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[35] Wikipedia, Binary search tree — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[36] Wikipedia, Tree (data structure) — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[37] Wikipedia, Binary tree — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[38] Wikipedia, AVL tree — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[39] W. contributors, Red–black tree — Wikipedia, The Free Encyclopedia, [Online; accessed 21-February-2018], **2018**.

[40] Wikipedia, Big O notation — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[41] lwn.net, Red-black trees, [Online; accessed 18-Apr-2015], **2006**.

[42] R. Landly, Red-black Trees (rbtree) in Linux, [Online; accessed 18-Apr-2015], **2007**.

[43] gcc.gnu.org, GCC Manual, [Online; accessed 18-Apr-2015], **2015**.

[44] ISO, ISO C Standard 1999, tech. rep., ISO/IEC 9899:1999 draft, ISO, **1999**.

[45] ibm.com, Designated initializers for aggregate types (C only), [Online; accessed 19-Apr-2015], **2015**.

[46] Wikipedia, Endianness — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[47] W. Swanson, The Art of Picking Intel Registers, http://www.swansontec.com/sregisters.html, (Accessed on 2016-06-26), **2003**.

[48] stackoverflow, What does `rep stos` do?, [Online; accessed 20-Apr-2015], **2010**.

[49] Wikipedia, FLAGS register — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[50] J. Bartlett, *Programming from the Ground Up*, University Press of Florida, **2009**.

[51] Wikibooks, GAS Syntax, **2012**.

[52] Wikipedia, Call stack — Wikipedia, The Free Encyclopedia, [Online; accessed 19-April-2015], **2015**.

[53] Wikipedia, Data structure alignment — Wikipedia, The Free Encyclopedia, [Online; accessed 25-April-2015], **2014**.

[54] ibiblio.org, GCC Inline Assembly HOWTO, **2012**.

[55] kernelnewbies.org, FAQ/asmlinkage, http://kernelnewbies.org/FAQ/asmlinkage, (Accessed on 2016-06-26), **2011**.

[56] Wikipedia, Calling convention — Wikipedia, The Free Encyclopedia, [Online; accessed 25-April-2015], **2015**.

[57] cprogramming.com, ((uint32_t)(~0UL)) ??, http://cboard.cprogramming.com/c-programming/130111-uint32$_t - \sim 0ul.html$, (Accessed on 2016-06-26), **2010**.

[58] Wikipedia, Volatile (computer programming) — Wikipedia, The Free Encyclopedia, [Online; accessed 25-April-2015], **2015**.

[59] P. J. Salzman, M. Burian, O. Pomerantz, *The Linux Kernel Module Programming Guide*, 2.6.4, tldp.org, **2007**.

[60] Wikipedia, Executable and Linkable Format — Wikipedia, The Free Encyclopedia, [Online; accessed 12-May-2015], **2015**.

[61] G. Duarte, Motherboard Chipsets and the Memory Map, **2008-06**.

[62] G. Duarte, How Computers Boot up, **2008**.

[63] stackoverflow, How is the BIOS ROM mapped into address space on PC?, [Online; accessed 20-Apr-2015], **2011**.

[64] www.pcguide.com, System Boot Sequence, [Online; accessed 20-Apr-2015], **2001**.

[65] stackoverflow, Do normal x86 or AMD PCs run startup/BIOS code directly from ROM, or do they copy it first to RAM?, [Online; accessed 20-Apr-2015], **2011**.

[66] Wikipedia, Reset vector — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2015**.

[67] lateblt.tripod.com, What Happens When A CPU Starts, [Online; accessed 20-Apr-2015].

[68] T. F. D. Project, FreeBSD Architecture Handbook, [Online; accessed 20-Apr-2015], **2013**.

[69] Wikipedia, Real mode — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2015**.

[70] wiki.osdev.org, Real Mode, [Online; accessed 20-Apr-2015], **2014**.

[71] wiki.osdev.org, Descriptor Cache, [Online; accessed 20-Apr-2015], **2009**.

[72] forum.osdev.org, Unreal mode, [Online; accessed 20-Apr-2015], **2009**.

[73] files.osdev.org, Segment Registers: Real mode vs. Protected mode, [Online; accessed 20-Apr-2015].

[74] Wikipedia, Unreal mode — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2015**.

[75] wiki.osdev.org, Unreal Mode, [Online; accessed 20-Apr-2015], **2014**.

[76] L. Benschop, Linux Boot Loaders Compared, [Online; accessed 20-Apr-2015], **2003**.

[77] K. source, The Linux Boot Protocol, **2008**.

[78] lists.kernelnewbies.org, From which point onwards the kernel execution starts?, [Online; accessed 20-Apr-2015], **2011**.

[79] U. Blog, Insight into GNU/Linux boot process, [Online; accessed 20-Apr-2015], **2010**.

[80] Sudhansu, Linux Boot Process in a nutshell, [Online; accessed 20-Apr-2015], **2012**.

[81] A. Nayani, M. Gorman, R. S. de Castro, *Memory Management in Linux: Desktop Companion to the Linux Source Code*, Free book, **2002**.

[82] Wikipedia, E820 — Wikipedia, The Free Encyclopedia, [Online; accessed 26-April-2015], **2013**.

[83] Wikipedia, A20 line — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2015**.

[84] Wikipedia, Interrupt descriptor table — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2014**.

[85] Wikipedia, Global Descriptor Table — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2014**.

[86] www.tldp.org, Linux 2.4.x Initialization for IA-32 HOWTO, [Online; accessed 20-Apr-2015], **2001**.

[87] Intel, INTEL 80386 Programmer's Reference Manual, **1986**.

[88] Wikipedia, X86 assembly language — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2015**.

[89] F. Wang, Linux i386 Boot Code HOWTO, [Online; accessed 20-Apr-2015], **2004**.

[90] 王爽, 汇编语言, 清华大学出版社, **2003**.

[91] G. Duarte, CPU Rings, Privilege, and Protection, [Online; accessed 20-Apr-2015], **2008**.

[92] G. Duarte, Memory Translation and Segmentation, [Online; accessed 20-Apr-2015], **2008**.

[93] Wikipedia, Task state segment — Wikipedia, The Free Encyclopedia, [Online; accessed 20-April-2015], **2014**.

[94] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley & Sons, **2008**.

[95] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Prentice Hall, **2004**.

[96] linux-mm.org, HighMemory, [Online; accessed 20-Apr-2015], **2006**.

[97] stackoverflow, Does there exist Kernel stack for each process ?, [Online; accessed 20-Apr-2015], **2011**.

[98] stackoverflow, kernel stack for linux process, [Online; accessed 20-Apr-2015], **2009**.

[99] stackoverflow, kernel stack vs user-mode application stack, [Online; accessed 20-Apr-2015], **2009**.

[100] U. for Dummies Questions & Answers, Kernel Stack vs User Mode Stack, [Online; accessed 20-Apr-2015], **2012**.

[101] www.linuxquestions.org, Kernel Stack Initialization, [Online; accessed 20-Apr-2015], **2011**.

[102] M. Bar, The Linux Process Model, [Online; accessed 20-Apr-2015], **2000**.

[103] Quora, What is the advantage of using zero-length arrays in C?, [Online; accessed 20-Apr-2015], **2012**.

[104] stackoverflow, Understanding the getting of task_struct pointer from process kernel stack, [Online; accessed 20-Apr-2015], **2012**.