Linux Kernel Analysis

Wang Xiaolin
wx672ster+os@gmail.com

July 26, 2020

OS Fundamentals

- TANENBAUM A S. *Modern Operating Systems*. 3rd ed. Prentice Hall Press, 2007.
- Silberschatz, Galvin, Gagne. *Operating System Concepts Essentials*. 1st ed. John Wiley & Sons, 2011.

Linux Kernel

- CORBET J, KROAH-HARTMAN G, RUBINI A. *Linux Device Drivers*. 3rd ed. O'Reilly, 2005.
- BOVET D, CESATI M. *Understanding The Linux Kernel*. 3rd ed. OReilly, 2005.
- MAUERER W. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008.
- LOVE R. *Linux Kernel Development*. Addison-Wesley, 2010.
- RODRIGUEZ C, FISCHER G, SMOLSKI S. *The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*. Prentice Hall Professional Technical Reference, 2005.

Assembly and C Programming

- BARTLETT J. *Programming from the Ground Up*. University Press of Florida, 2009.
- CARTER P. *PC Assembly Language*. www.drpaulcarter.com, 2006.
- Oracle. *X86 Assembly Language Reference Manual*. http://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf. 2012.
- Wikibooks. *GAS Syntax*. http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax. 2012.
- Ibiblio.org. *GCC Inline Assembly HOWTO*. http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html. 2012.
- DEVELOPERWORKS I. *Inline assembly for x86 in Linux*. http://www.ibm.com/developerworks/linux/library/l-ia/index.html. 2012.
- Stackoverflow. *Understanding gcc generated assembly*. http://stackoverflow.com/questions/8478491/understanding-base-pointer-and-stack-pointers-in-context-with-gcc-output. 2012.

Lab References

- KROAH-HARTMAN G. *Linux Kernel in a Nutshell: A Desktop Quick Reference*. O’Reilly, 2007.
- 郑钢. 操作系统真象还原. 人民邮电出版社, 2016.
- 新设计团队. *Linux* 内核设计的艺术. 机械工业出版社, 2013.
- 于渊. *Orange’S*：一个操作系统的实现. 电子工业出版社, 2009.

Homework

Weekly tech question

1. What was I trying to do?
2. How did I do it? (steps)
3. The expected output? The real output?
4. How did I try to solve it? (steps, books, web links)
5. How many hours did I struggle on it?

✉ `wx672ster+os@gmail.com`

$\mathbb{E}$ Preferably in English

in stackoverflow style

OR simply show me the tech questions you asked on any website

1 Overview

1.1 Basic Operating System Concepts

Basic Operating System Concepts

Two main objectives of an OS:

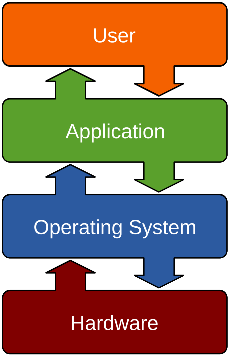
- Interact with the hardware components
- Provide an execution environment to the applications

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



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

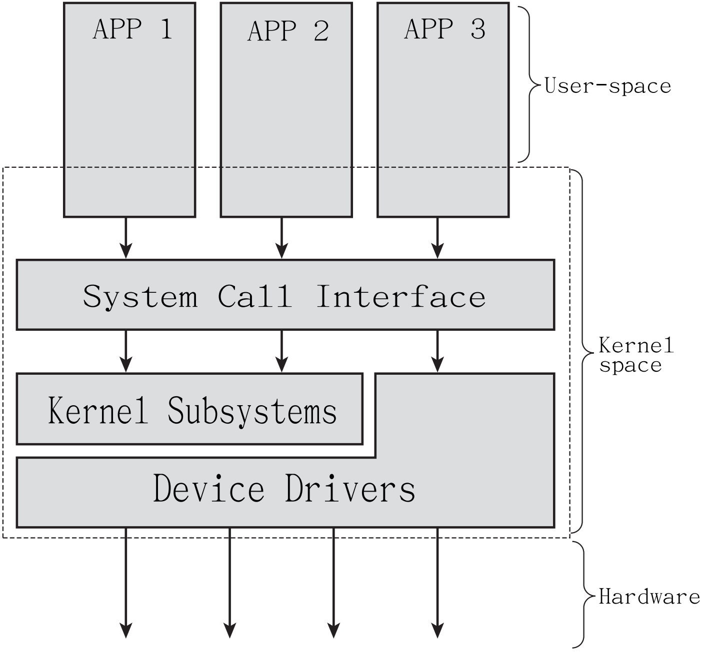
When executing the kernel, the system is in kernel-space executing in *kernel mode*.

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*.

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


APP 1
APP 2
APP 3
User-space
System Call Interface
Kernel Subsystems
Device Drivers
Kernel
space
Hardware

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

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 $\Leftrightarrow$ 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.

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

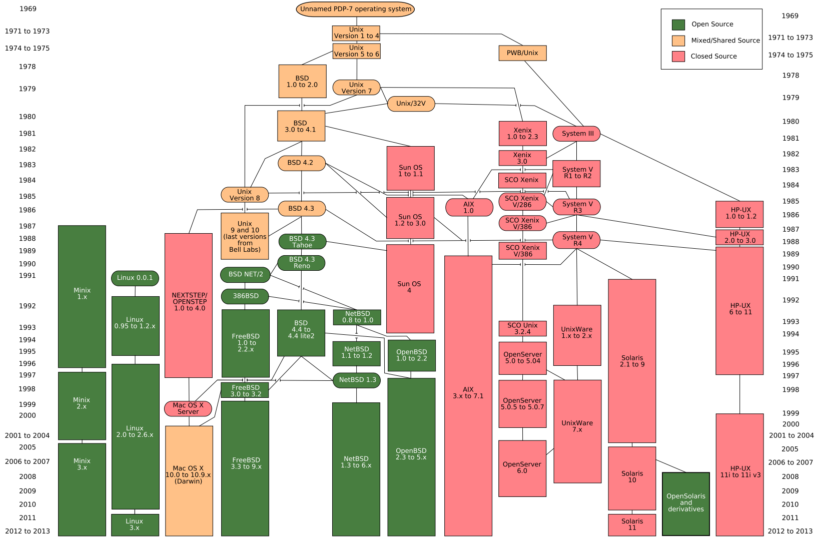
Platform independence — if standards compliant

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



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 ...

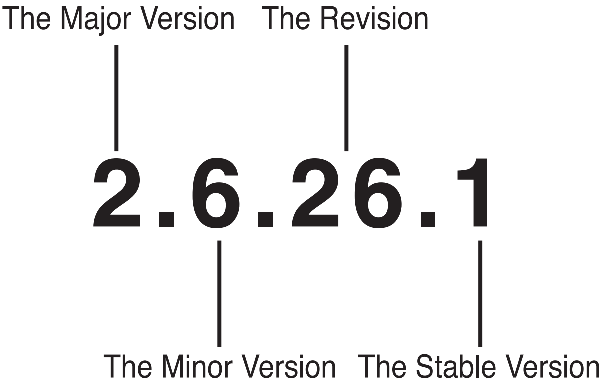
Linux Kernel Features

- 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

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

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

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

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

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

```
+------+          +------+                           +------+
| User |          | User |                           | User |
+------+↓        ↑+------+↓                         ↑+------+    user mode
--------↓--------↑--------↓-------------------------↑----------------------
        ↓+------+↑        ↓+------+         +------+↑               kernel mode
         | Excp |          | Intr |         | Intr |
         +------+          +------+↓       ↑+------+
                                   ↓       ↑
                                   ↓+------+↑
                                    | Intr |
                                    +------+
➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ ➤ Time ➤ ➤ ➤ ➤
```

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()`

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

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
```

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

Zombie processes

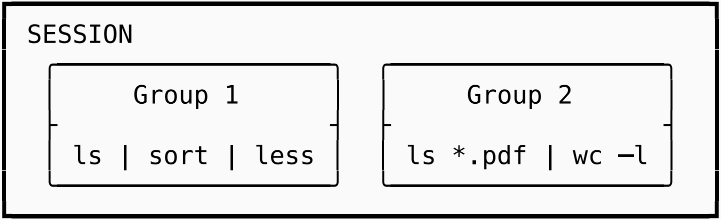
Zombie a *process state* representing terminated processes

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

Orphaned processes become children of *init*.

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

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

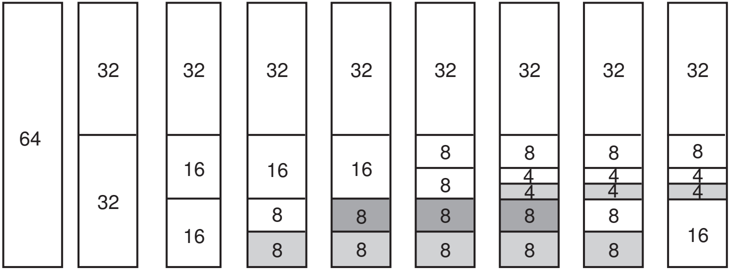
Because:

- 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*.

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

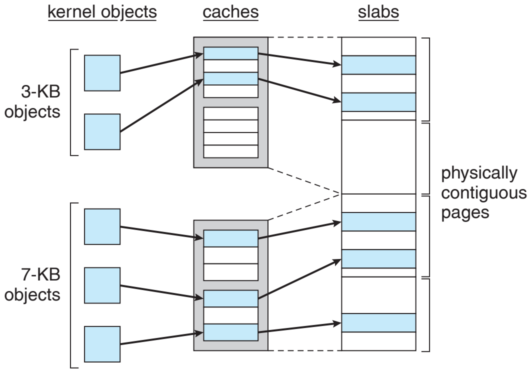
- No memory is wasted due to fragmentation
- Memory request can be satisfied quickly

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()`

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

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.

2.2 A Beast Of A Different Nature

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

Inline assembly

Embedding assembly instructions in normal C functions

- ☺ speed
- ☹ Architecture dependent (poor potability)

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

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

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;
}
```

No memory protection

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

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

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

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`

Common Kernel Data-types

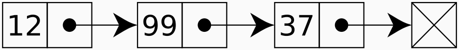
Many objects and structures in the kernel

- memory pages
- processes
- interrupts
- ...

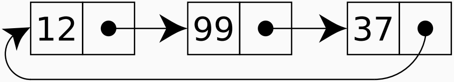
1. linked-lists — to group them together
2. binary search trees — to efficiently find a single element

Linked Lists

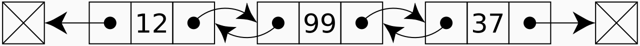
Singly linked list



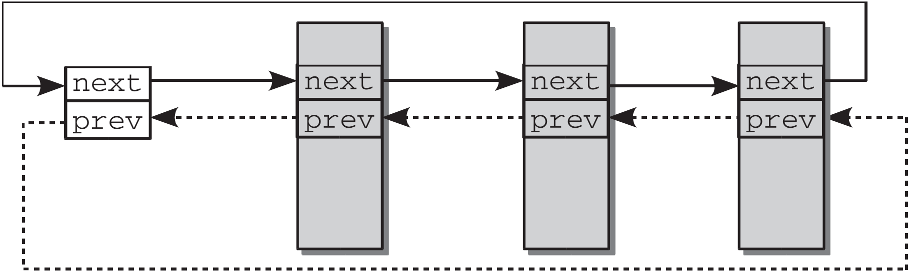
Circularly linked list



Doubly linked list



Circular doubly linked-lists



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`?

Example

```c
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

```c
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);
```

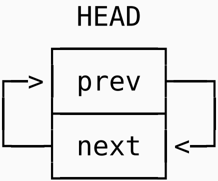
- Compile-time initialization

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

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);
```

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...

List Traversing

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

list_for_each

```c
/**
 * 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.

```c
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*?

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

list_entry(ptr, type, member)

```c
/**
 * 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)
```

list_for_each_entry(pos, head, member)

```c
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)

```c
/**
 * 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))
```

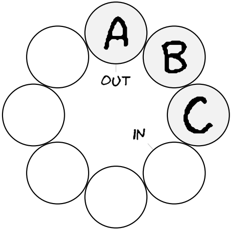
Queue (FIFO)

Circular buffer

Empty: $in == out$

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

Can be lock-free.



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;
```

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);
```

Example

1. To enqueue 32 integers into $fifo$

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

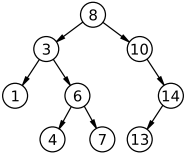
2. To dequeue and print all the items in the queue

```
/* 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);
}
```

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



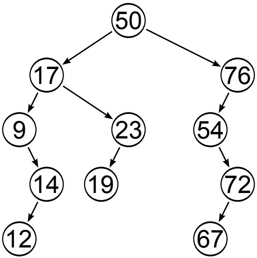
Properties:

- 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.

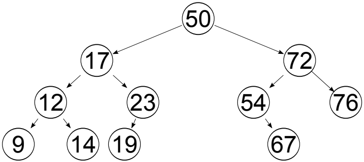
Efficient in:

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

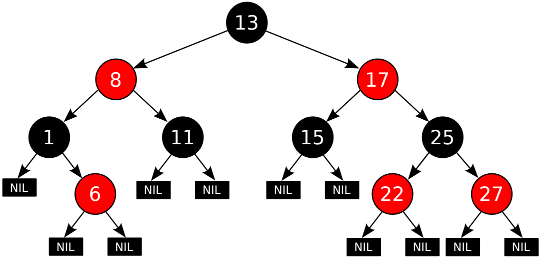
Unbalanced binary tree

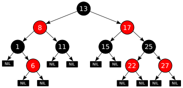


Balanced binary tree



Red-black tree





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*.

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

<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;
```

Example

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

Search

```c
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;
}
```

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

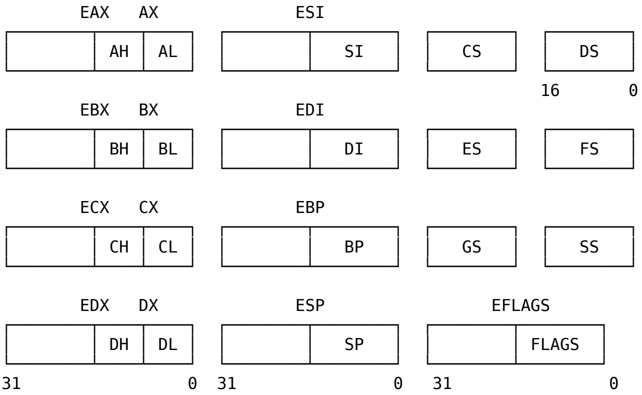
Storing an int (`0x01234567`) at address `0x100`:

Big endian

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

Little endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...



Three kinds of registers

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

General purpose registers

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

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

flag

ZF Zero flag

SF Sign flag, Negative

OF Overflow flag

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

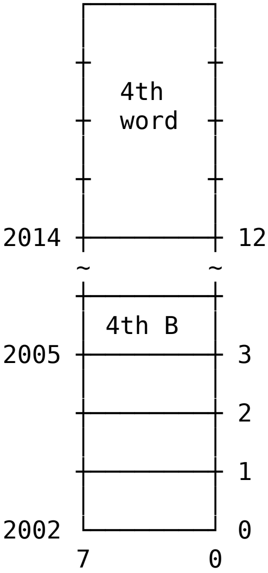
`START(,INDEX,MULTIPLIER)`:

- to access the 4th byte from location 2002

$$2002(,3,1) = 2002 + 3 \times 1 = 2005$$

- to access the 4th word from location 2002

$$2002(,3,4) = 2002 + 3 \times 4 = 2014$$



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

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
```

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
```

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.

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.

Generating Assembly From C Code

simple.c

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

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

simple.s (AT&T syntax)

```
        .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
```

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
```

Operation Prefixes

- **$** constant numbers
- **%** register

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)

Generating Assembly From C Code

count.c

```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
```

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
```

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" :::);
```

Example: `exit(0)`

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

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

2.5 Quirky C Language Usage

Quirky C Language Usage

`asmlinkage` and `fastcall`

asmlinkage

```c
asmlinkage int sys_fork(struct pt_regs regs)
```

- tells the compiler to pass parameters on the local stack.

fastcall

```c
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)))`

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 */
```

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)
```

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;
```

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`

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

```
$ hd a.out
```

nm List symbols from object files

```
$ nm a.out
```

2.8 Kernel Speaks: Listen to Kernel Messages

Listen To Kernel Messages

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

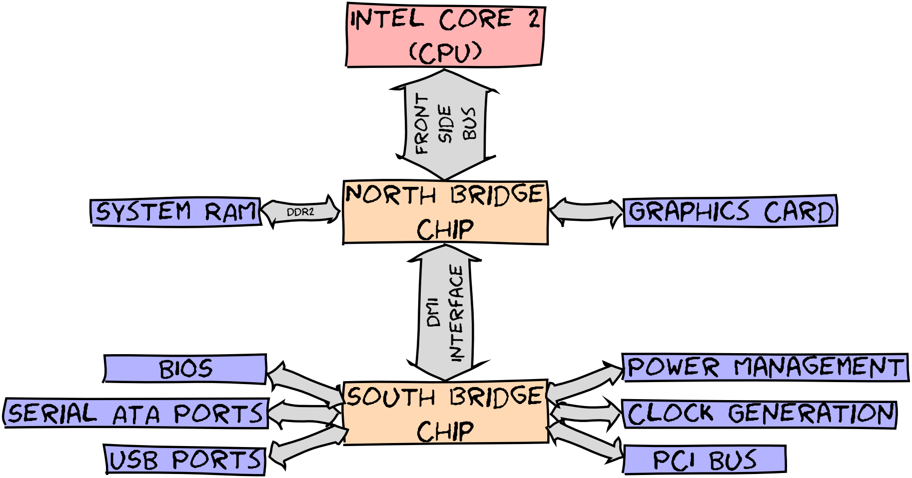
`dmesg` print or control the kernel ring buffer

`/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

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

Intel Core 2 QX6600

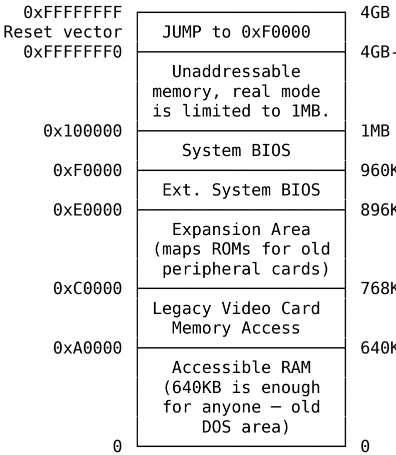
- 33 pins to transmit the physical memory address
  - so there are 2^{33} choices of memory locations
- 64 pins to send or receive data
  - so data path is 64-bit wide, or 8-byte chunks

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

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?

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

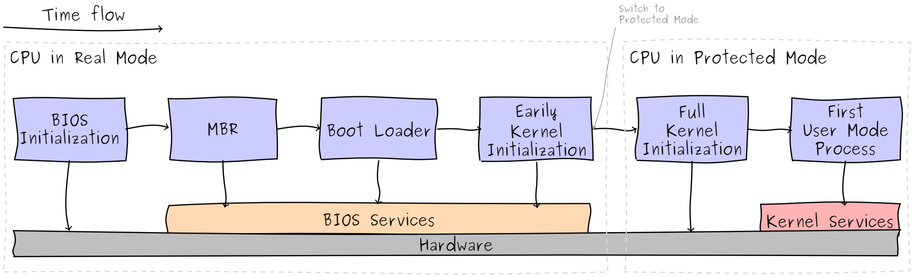
64-bit protected mode: can address 64GB RAM (Intel Core 2 QX6600)

- 33 address pins, 8-byte data unit

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

3.2 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`

Motherboard power up

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

Real mode

CPU acts as a 1978 Intel 8086

- 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`).

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

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:

$$\mathit{FFFF} \times 16 + 0001 = \mathit{FFFF}0 + 0001 = \mathit{FFFF}1$$

if: `offset > 0xF` (overflow)

then: $\texttt{address}\%2^{20}$ (wrap around)

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

$$\mathit{FFFF}0 + \mathit{FFFF} = 10\mathit{FFEF}$$

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
```

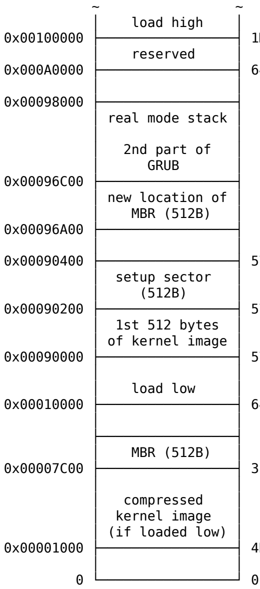
GRUB

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`

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



3.3 The Kernel Boot process

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()`

`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 ($\texttt{0x10000} \ll 4$)
  - `startup_32()` for real kernel

`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()`

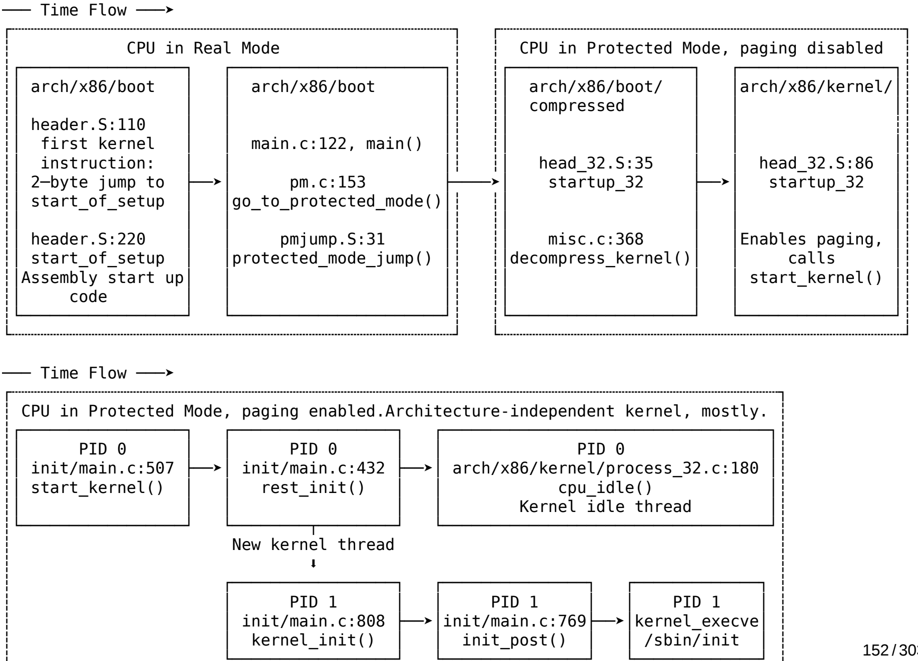
`start_kernel()` — a long list of calls to initialize various kernel subsystems and data structures

- `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

```
BIOS → Bootloader → setup()     → startup_32() → startup_32() → start_kernel()
                    PE=1;PG=0                    PG=1
```

The Kernel Boot Process

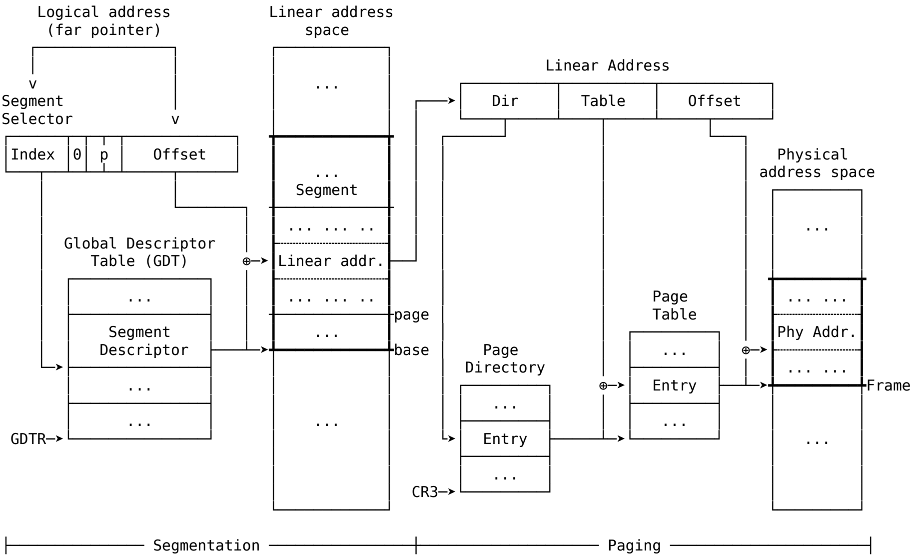


4 Memory Addressing

4.1 Memory Addresses

Three Kinds Of Addresses

```
          Logical     ┌──────────────┐   Linear      ┌────────┐  Physical     ┌──────────┐
 ┌─────┐  address     │ Segmentation │   address     │ Paging │  address      │ Physical │
 │ CPU │ ──────────>  │     unit     │ ──────────>   │  unit  │ ──────────>   │  memory  │
 └─────┘              └──────────────┘               └────────┘               └──────────┘
                    |<──────────────── MMU ────────────────>|
```


Logical address
(far pointer)
Segment
Selector
Index
0
p
Offset
Linear address
space
...
Segment
... .
Linear addr.
... .
page
...
base
Global Descriptor
Table (GDT)
...
Segment
Descriptor
...
...
GDTR
Linear Address
Dir
Table
Offset
Page
Directory
...
Entry
...
CR3
Page
Table
...
Entry
...
Physical
address space
...
... ...
Phy Addr.
... ...
Frame
...
Segmentation
Paging

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>`

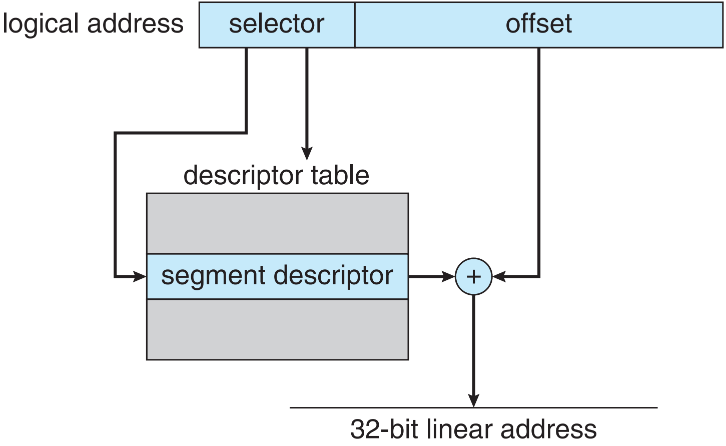
Real mode address translation

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

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

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

Protected Mode Address Translation

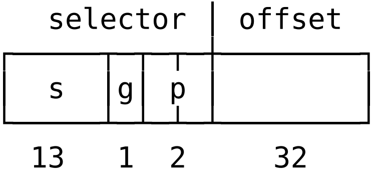


Segment Selectors

A logical address consists of two parts:

segment selector	:	offset
16 bits		32 bits

Segment selector is an index into GDT/LDT




```
s — segment number
g — 0 (global)
    1 (local)
p — protection use
```

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

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

Example: A LDT entry for code segment

Base 24-31	G	D	0		Limit 16-19	P	DPL	S	Type	Base 16-23	4
Base 0-15						Limit 0-15					0

← 32 Bits → Relative address

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

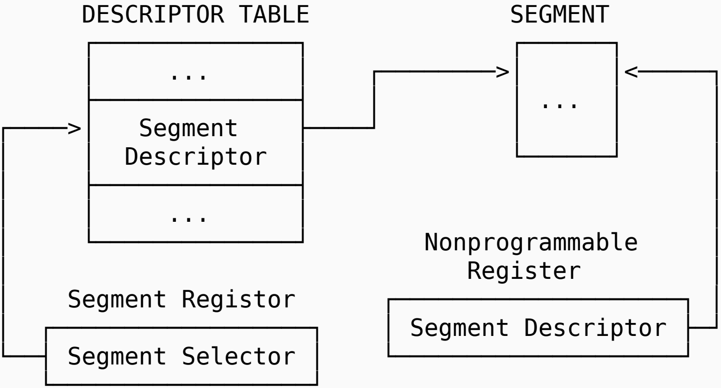
P: Segment-Present flag

- 0 - not in memory
- 1 - in memory

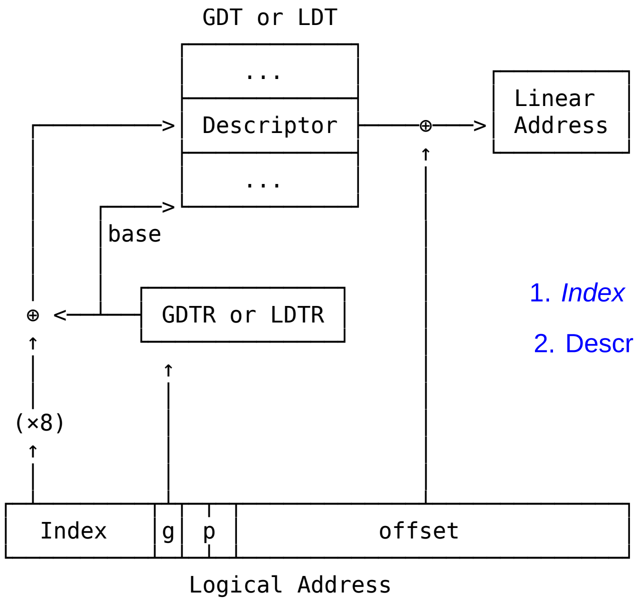
AVL: ignored by Linux

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

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

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

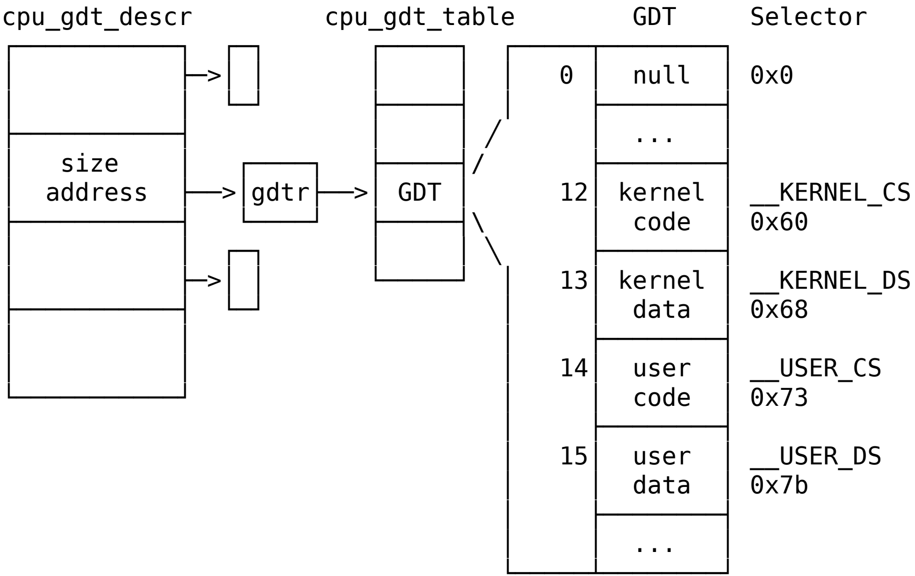
The Linux GDT

8 byte segment descriptor

	31 … 24	23			20	19 … 16	15		… 8	7 … 0	
7	BASE 31..24	1	1	0	0	LIMIT	1	DPL	TYPE	BASE 23..16	4
3	SEGMENT BASE 15..0						SEGMENT LIMIT 15..0				0

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 */
```



cpu_gdt_table: store 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

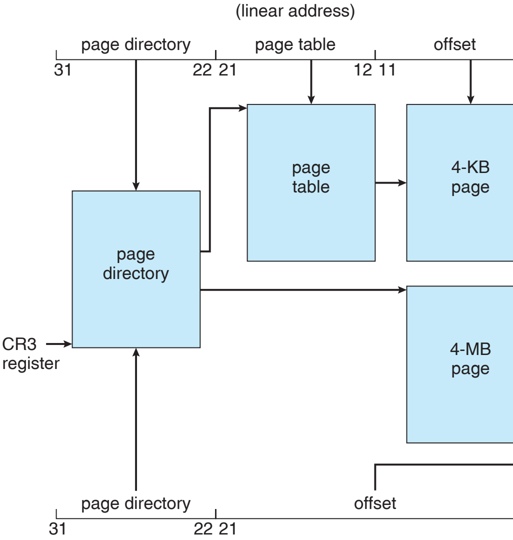
Linear Address ⇒ Physical Address

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
```



Same structure for Page Dirs and Page Tables

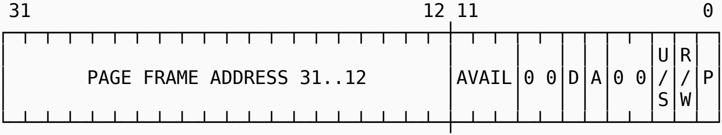
- 4 bytes (32 bits) long
- Page size is usually 4k (2^{12} bytes). OS dependent

  $ 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$\Rightarrow$ 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` $\times 4$

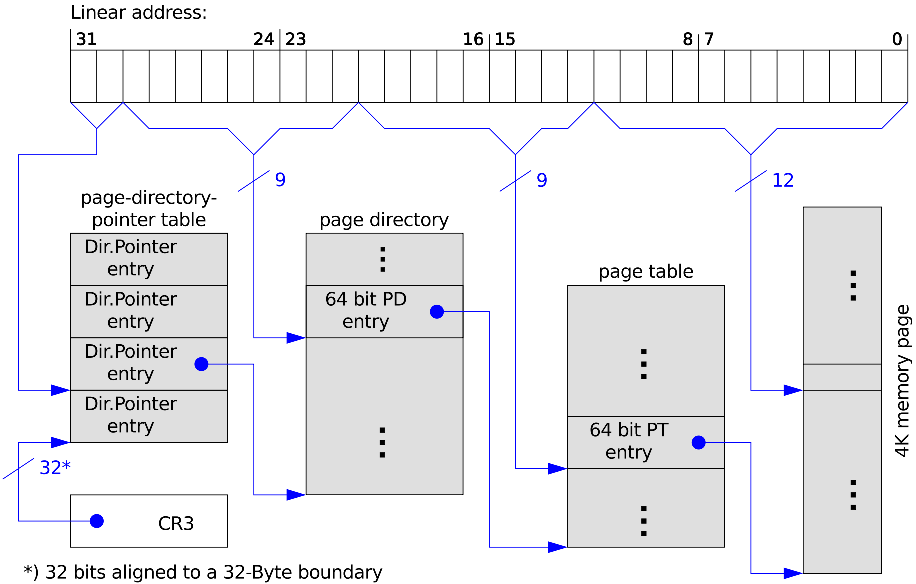
3–level paging for 4K–pages

PD PT	Page DIR	Page Table	Offset
2	9	9	12

2–level paging for 2M–pages

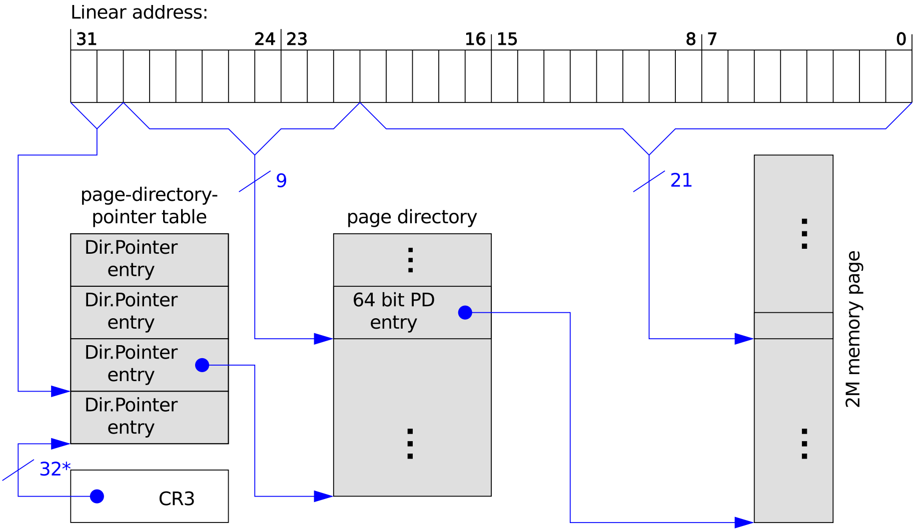
PD PT	Page DIR	Offset
2	9	21

PAE with 4K pages



*) 32 bits aligned to a 32-Byte boundary

PAE with 2M pages



*) 32 bits aligned to a 32-Byte boundary

Physical Address Extension (PAE)

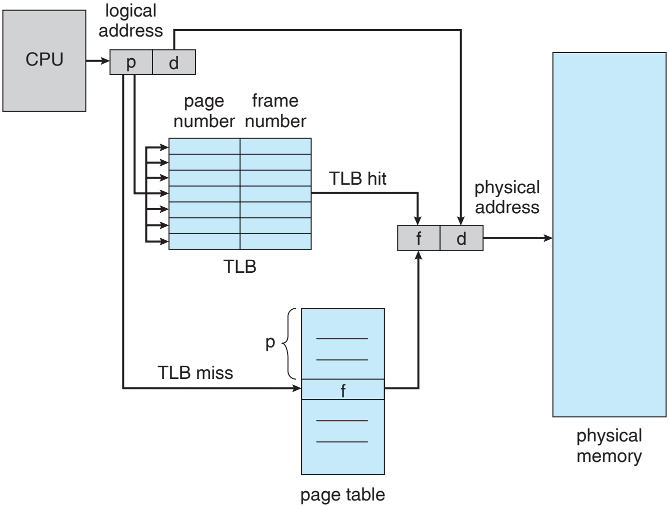
The linear address are still 32 bits

- 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

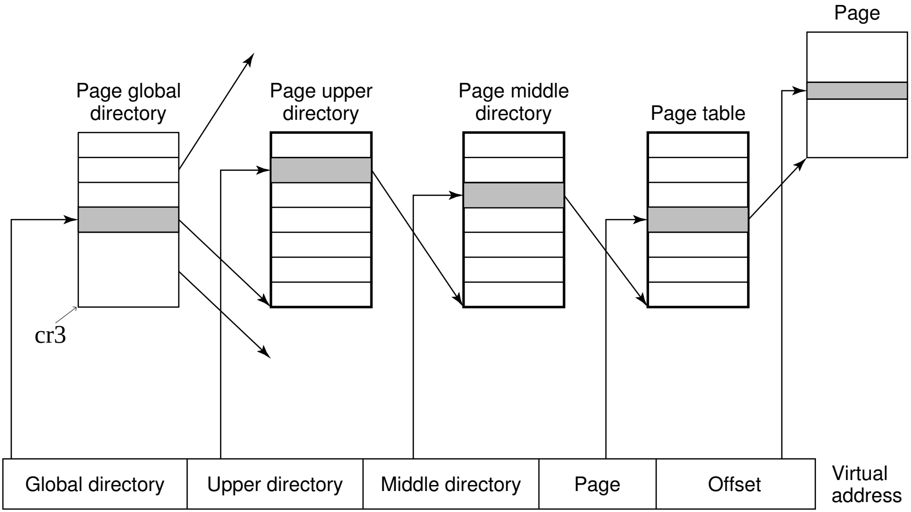
- Only a small fraction of the PTEs are heavily read; the rest are barely used at all



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	$9+9+9+\ \ 9+12$

The Linear Address Fields

```
|<-PGDIR_MASK->|<----------------PGDIR_SIZE---------------->|
|<-------PUD_MASK------->|<------------PUD_SIZE------------>|
|<------------PMD_MASK------------>|<-------PMD_SIZE-------->|
|<----------------PAGE_MASK---------------->|<-PAGE_SIZE->|
|                                           |             |
+--------------+----------+----------+----------+-------------+
| Global       | Upper    | Middle   | Page     | Offset      |
| DIR        ? | DIR    ? | DIR    ? | Table  ? |          12 |
+--------------+----------+----------+----------+-------------+
                                                |<-PAGE_SHIFT->|
                                     |<---PMD_SHIFT----------->|
                          |<-----PUD_SHIFT-------------------->|
               |<-----PGDIR_SHIFT----------------------------->|
```

- `*_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` include/asm-i386/page.h

```c
/* 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

```c
/*
 * 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

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

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

3–level paging for 4K–pages

PD PT	Page DIR	Page Table	Offset
2	9	9	12

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
```

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
```

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;
```

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	System BIOS	1MB
0xF0000	Ext. System BIOS	960KB
0xE0000	Expansion Area (maps ROMs for old peripheral cards)	896KB
0xC0000	Legacy Video Card Memory Access	768KB
0xA0000	Accessible RAM (640KB is enough for anyone – old DOS area)	640KB
0		0

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`

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

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

```
 page           160             256                                   768
frame: 0 1      0xa0            0x100                                0x300
       ┌─┬───────┬───────┬───────┬────────┬─────────────┬──────┬───────┬──
       │ │ avail │ resvd │ avail │ kernel │ Initialized │ BSS  │ avail │..
       │ │       │       │       │  code  │    data     │ data │       │..
       └─┴───────┴───────┴───────┴────────┴─────────────┴──────┴───────┴──
       0 4K     640K          _text    _etext        _edata   _end     3M
```

Process Page Tables

`0xC0000000` $\Leftrightarrow$ `PAGE_OFFSET`

include/asm-i386/page.h

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

	PAGE_OFFSET	
	user or kernel mode	kernel mode
0	3G	4G

Why?

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

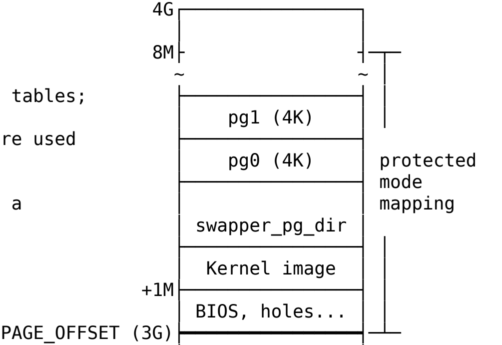
$$\textit{Physical} = \textit{Virtual} - \texttt{PAGE\_OFFSET}$$

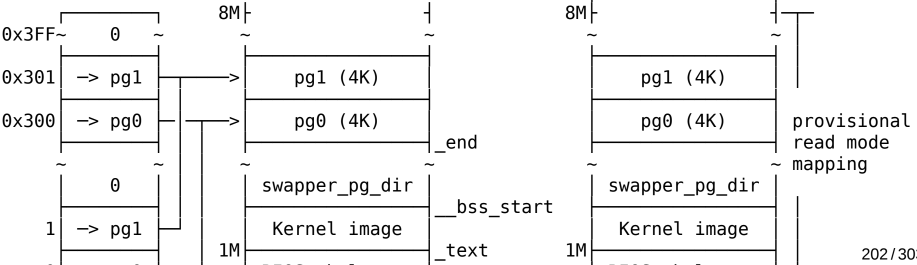
Kernel Page Tables

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.

Provisional Page Table Initialization

arch/i386/kernel/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   # 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)
```

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;
    }
  }
```

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();
}
```

2 level paging: `PUD` and `PMD` are folded

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

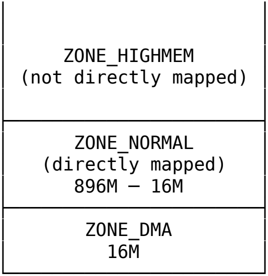
`pagetable_init()` — re-initializes the `PGDir` at `swapper_pg_dir`

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;
}
```

When RAM Size Is Between 896MB $\sim$ 4096MB

Physical memory zones:

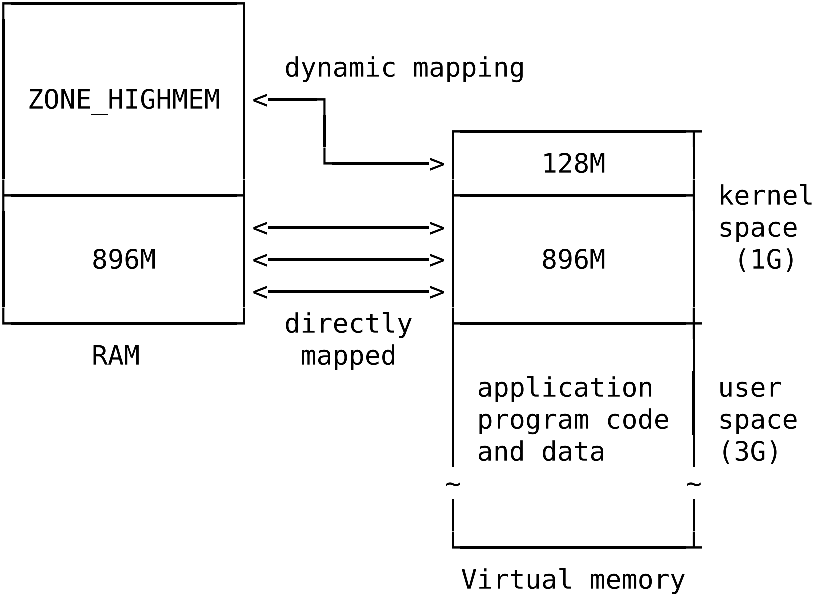


Physical RAM

Direct mapping for `ZONE_NORMAL`:

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

High Memory


ZONE_HIGHMEM
dynamic mapping
128M
kernel space (1G)
896M
896M
directly mapped
RAM
application program code and data
user space (3G)
Virtual memory

When RAM Size Is More Than 4096MB (PAE)

A 3-level paging model is used

3–level paging for 4K–pages

PD PT	Page DIR	Page Table	Offset
2	9	9	12

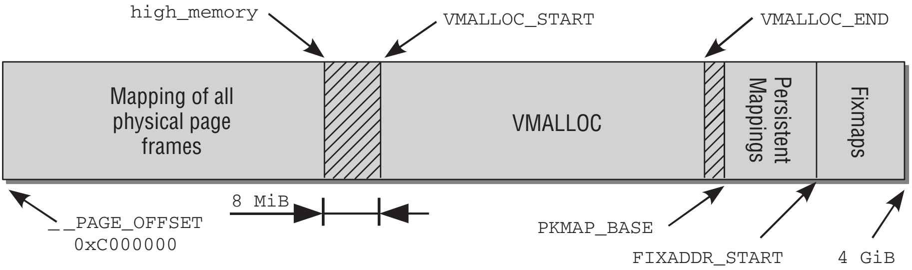
PGDir	PUD	PMD	PT	OFFSET
2	0	9	9	12

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

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */
for (i=0; i<pgd_idx; i++)
  set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
  /* 0x001 == Present */
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
  pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
  set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */
  if (phys_addr < max_low_pfn * PAGE_SIZE)
    for (j=0; j < PTRS_PER_PMD /* 512 */
      && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
      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; /* 0x200000 */
    }
 }
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

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.

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

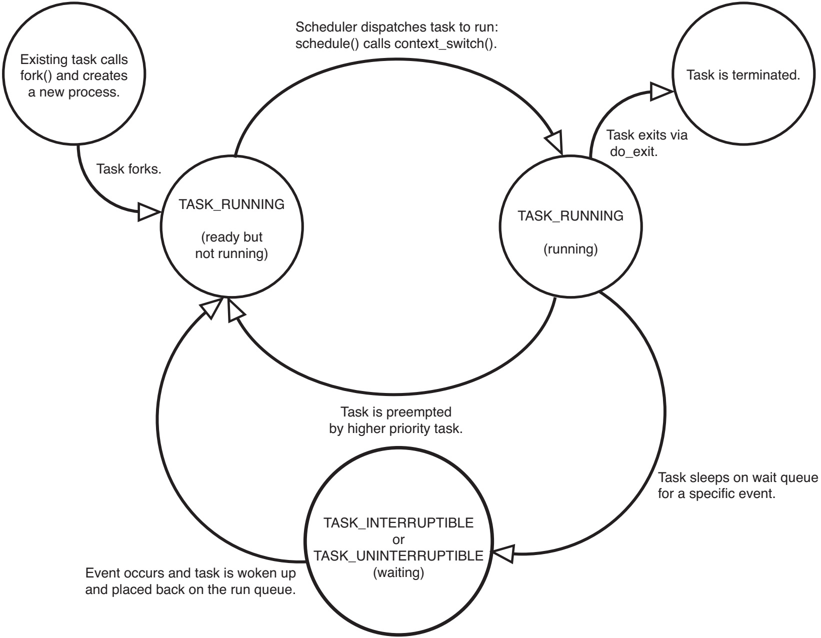
To manage processes, the kernel must have a clear picture of what each process is doing.

- 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 */
};
```

Process State



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
```

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?

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

`pidmap_array` consumes a single page.

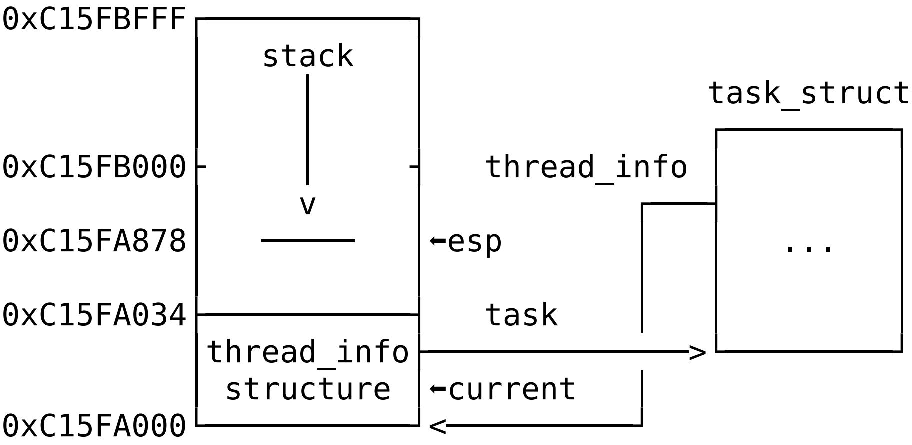
Process Descriptor Handling

`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 */
};
```

Kernel Stack



```
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

thread_info and task_struct are mutually linked

```
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()`

```
/* 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
```

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.

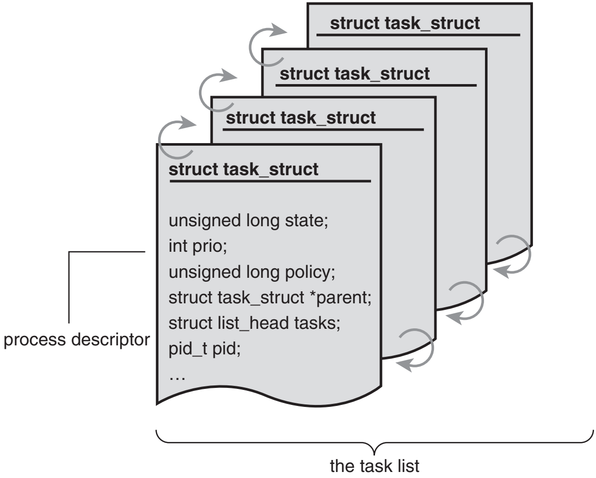
`current` — a marco pointing to the current running task

```
static inline struct task_struct * get_current(void)
{
        return current_thread_info()->task;
}

#define current get_current()
```

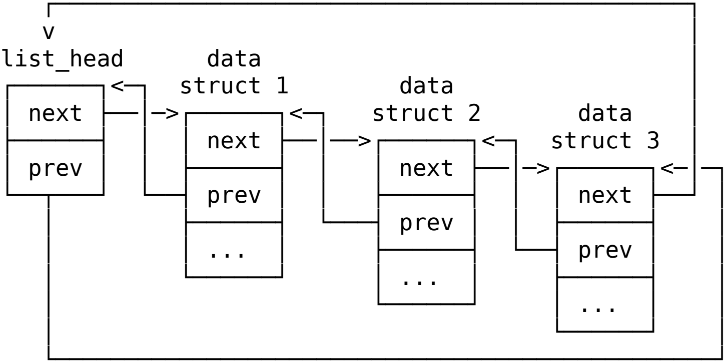
Task List

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.



Doubly Linked List




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

```c
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

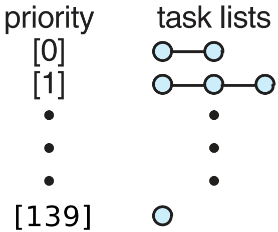
```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
  task = list_entry(list, struct task_struct, sibling);
  /* task now points to one of current's children */
}
```

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




```c
struct task_struct {
  ...
  int prio, static_prio;
  struct list_head run_list;
  prio_array_t *array;
  ...
};
```

Each Runqueue Has A `prio_array_t` Struct

```c
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

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.

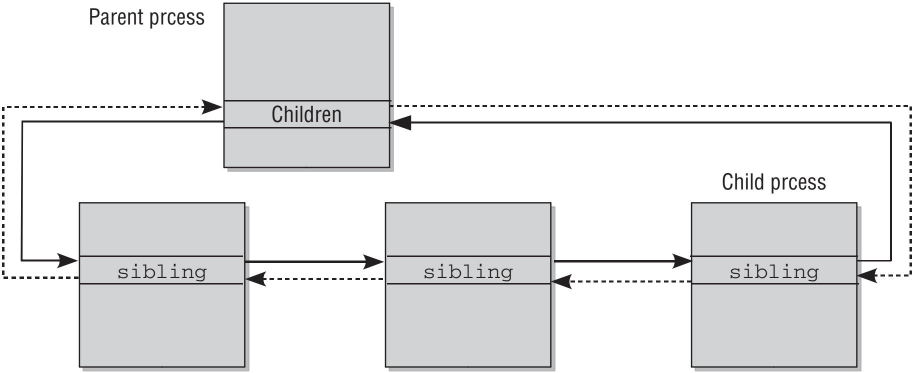
Relationships Among Processes

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


Parent prcess
Children
Child prcess
sibling
sibling
sibling

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

```c
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 $\Rightarrow$ process descriptor pointer?

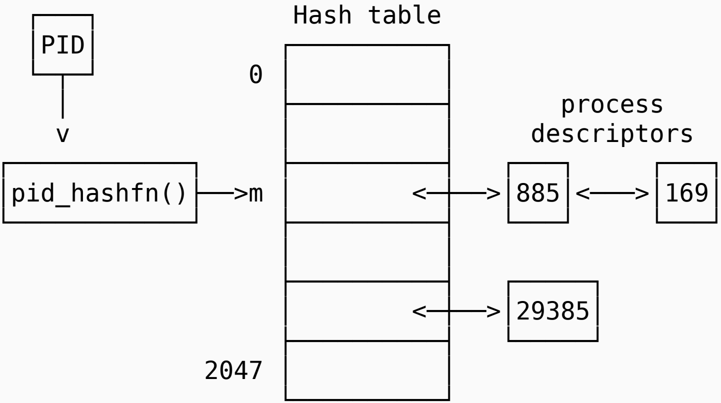
- Scanning the process list? — too slow
- Use hash tables

Four hash tables have been introduced

Why 4? For 4 types of PID $\left.\begin{array}{r}\texttt{PID}\\ \texttt{TGID}\\ \texttt{PGID}\\ \texttt{SID}\end{array}\right\} \Rightarrow \texttt{task\_struct}$

Collision

Multiple PIDs can be hashed into one table index



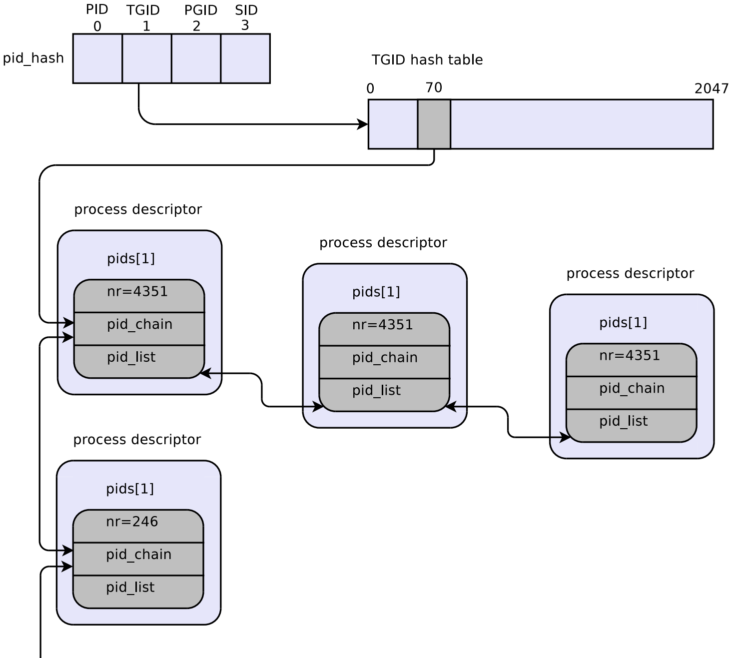
- Chaining is used to handle colliding PIDs
- No collision if the table is 32768 in size! But...

The `pid` 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)`

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?

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 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.

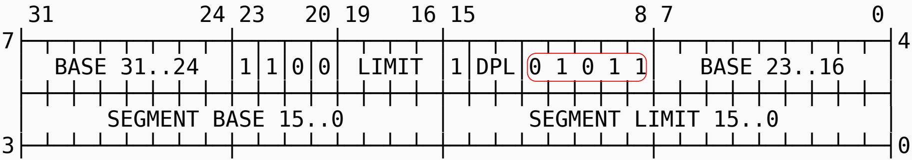
Task State Segment (TSS)

- 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

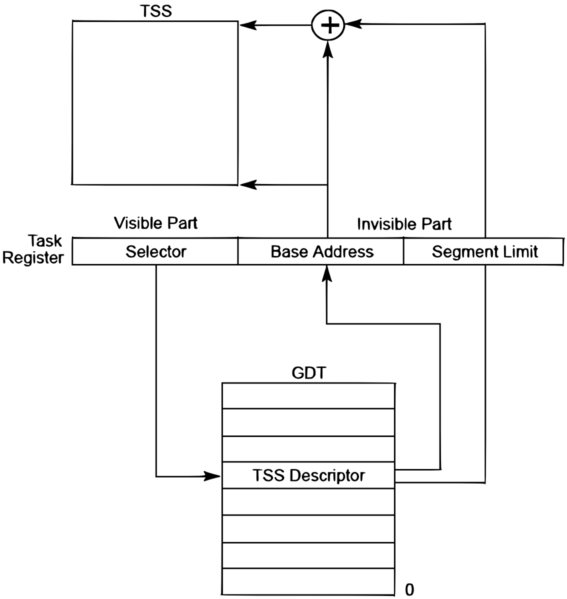
- One TSS for each CPU
  - The address of the kernel mode stack
  - I/O permission bitmap

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

5.4 Creating Processes

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);
```

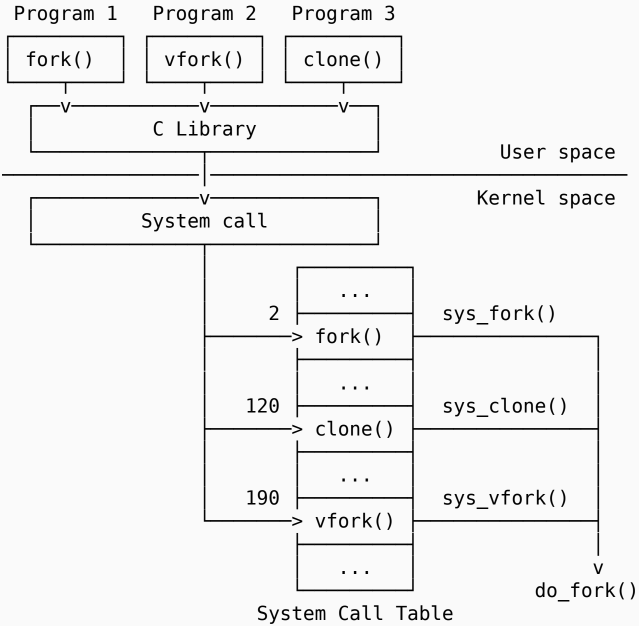
- `child_stack`: parent stack pointer (copy-on-write)

`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;

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);
}
```

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

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

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

6.3 Scheduling Policy

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 $\Rightarrow$ lower priority
   - lower value $\Rightarrow$ higher priority $\Rightarrow$ 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 $\Rightarrow$ 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.

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

CFS

In practice it works like this:

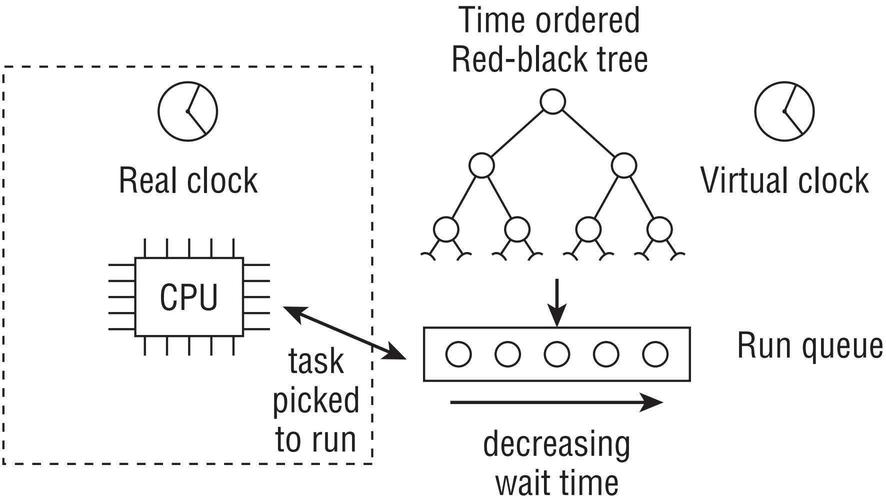
- While a task is using the CPU, its `wait_runtime` decreases

$$\texttt{wait\_runtime} = \texttt{wait\_runtime} - \texttt{time\_running}$$

if: its $\texttt{wait\_runtime} \neq \texttt{MIN}_{\texttt{wait\_runtime}}$ (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

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

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)

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

6.5 The Linux Scheduling Implementation

Base Time Quantum

$O(1)$ scheduler

$$\text{base time quantum (ms)} = \begin{cases} (140 - \text{static priority}) \times 20 & \textit{if } \text{static priority} < 120, \\ (140 - \text{static priority}) \times 5 & \textit{if } \text{static priority} \geq 120. \end{cases}$$

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()`

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()`

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