Before Diving Into The Kernel Source

Wang Xiaolin
wx672ster@gmail.com

June 18, 2013

Contents

References

[Bar09] J. Bartlett. *Programming from the Ground Up*. University Press of Florida, 2009.

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[CKR05] Jonathan Corbet, Greg Kroah-Hartman, and Alessandro Rubini. *Linux Device Drivers*. 3rd ed. O'Reilly, 2005.

[ibi12] ibiblio.org. *GCC Inline Assembly HOWTO*. 2012.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Addison-Wesley, 2010.

[RFS05] C.S. Rodriguez, G. Fischer, and S. Smolski. *The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*. Prentice Hall Professional Technical Reference, 2005.

[SBP07] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. 2.6.4. tldp.org, 2007.

[Wik12] Wikibooks. *GAS Syntax*. 2012.

1 Getting Started with the Kernel

Textbook: Chapter 2 of [Lov10]

Obtaining The Kernel Source

Web: 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

- Defaults: make defconfig
- Using .config file: make oldconfig

Make

```
$ make > /dev/null
$ 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

- Debian Linux Kernel Handbook
- Compile Linux Kernel on Ubuntu 12.04 LTS (Detailed)

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 is to place inline functions in header files

---

- www.greenend.org.uk: Inline Functions In C
- Wikipedia: Inline function

Inline assembly
Embedding assembly instructions in normal C functions

- Architecture dependent
- speed

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

unsigned int low, high;

asm volatile(”rdtsc” : ”=a” (low), ”=b” (high));

- Wikipedia: Inline assembly

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;
}
```

(Sec 2.8.2 in [RFS05]) In this code, we see that a syscall to get the time of day is likely to have a timeval structure that is not null. If it were null, we couldn't fill in the requested time of day! It is also unlikely that the timezone is not null. To put it another way, the caller rarely asks for the timezone and usually asks for the time.

More info:

- Kerneltrap.org: likely/unlikely macros
- Kernelnewbies.org: FAQ/LikelyUnlikely

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

No memory protection

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

- Wikipedia: Linux kernel oops

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

- Stackoverflow.com: Use of floating point in the Linux kernel
- Linux Kernel and Floating Point

Small, fixed-size stack

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

3 Common Kernel Data-types

Textbook: Chapter 2 in [RFS05]; Chapter 6 in [Lov10]; Chapter 11 in [CKR05]

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

(p289, [CKR05]) Although you must be careful when mixing different data types, sometimes there are good reasons to do so. One such situation is for memory addresses, which are special as far as the kernel is concerned. Although, conceptually, addresses are pointers, memory administration is often better accomplished by using an unsigned integer type; *the kernel treats physical memory like a huge array, and a memory address is just an index into the array.* Furthermore, a pointer is easily dereferenced; when dealing directly with memory addresses, you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs. Therefore, *generic memory addresses in the kernel are usually unsigned long, exploiting the fact that pointers and long integers are always the same size, at least on all the platforms currently supported by Linux.*

Common Kernel Data-types

Many objects and structures in the kernel

- memory pages
- processes
- interrupts
- ...

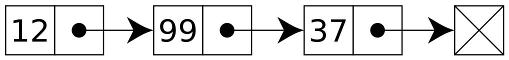
1. linked-lists — to group them together
2. binary search trees — to efficiently find a single element

3.1 Linked Lists

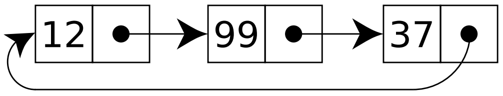
Textbooks: ch 11 in [CKR05], sec 2.1 in [RFS05]

Linked Lists

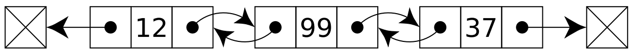
Singly linked list



Circularly linked list

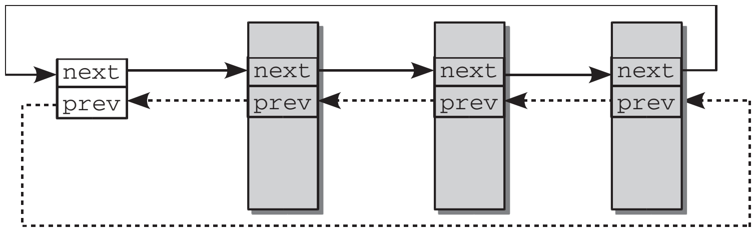


Doubly linked list



- Linked lists are ill-suited for use cases where *random access* is an important operation. Instead, you use linked lists when *iterating over the whole list* is important and the *dynamic addition and removal* of elements is required (Sec 6.1.3 in [Lov10]).
- stackoverflow.com: Array vs. linked list.

Circular doubly linked-lists



More info:

- sec 3.2.2.3 in [BC05]
- sec 2.1 in [RFS05]
- `http://kernelnewbies.org/FAQ/LinkedLists`
- Linux Kernel Linked List Explained

Things to note

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

- Linux Kernel Linked List Explained

Linked Lists
A linked list is initialized by using the LIST_HEAD and INIT_ LIST_HEAD macros

include/linux/list.h

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

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

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

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

Q1: Why both LIST_HEAD_INIT and INIT_ LIST_HEAD?

Q2: Why do-while?

LIST_HEAD Quoted from [`http://kernelnewbies.org/FAQ/LinkedLists`]:

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

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

A:

- See Sec *Defining a linked list* in [Lov10] for an example.
- Quote from [ubuntuforums.org: link list implementation in linux kernel easy question syntax not clear]:

  > LIST_HEAD_INIT initializes one of these list head structure thingamajigs at compile time and so it doesn't actually generate any code, but stores constant values in a structure of which the address is known and fixed.
  > OTOH INIT_LIST_HEAD takes a pointer and so it could be a structure that is dynamically allocated, e.g. with "malloc" or one that is passed in to your function... and then it does it's stuff at run time.

- Quote from [Stackoverflow: Linux Kernel List LIST_HEAD_INIT vs INIT_LIST_HEAD]:

  > LIST_HEAD_INIT is a static initializer, INIT_LIST_HEAD is a function (in 2.6.11 it's still a macro). They both initialise a list_head to be empty.
  > If you are statically declaring a list_head, you should use LIST_HEAD_INIT, eg:
  >
  > static struct list_head mylist = INIT_LIST_HEAD(mylist);
  >
  > You should use INIT_LIST_HEAD() for a list head that is dynamically allocated, usually part of another structure. There are many examples in the kernel source.

- **My understanding:** compile time list head initialization is better for kernel memory management e.g. slab allocation.
- Wikipedia: C dynamic memory allocation rationale

Example

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

The list needs to be initialized before in use

- Run-time initialization

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

- Compile-time initialization

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

- Sec 6.1.4 in [Lov10]
- About kmalloc(), see sec 8.1 in [CKR05]
- linux-mm.org: GFP MASK

The do while(0) trick

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

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

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

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

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

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

do...while(0) Quoted from [ubuntuforums.org: link list implementation in linux kernel easy question syntax not clear]:

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

More info:

- stackoverflow: What's the use of do while(0) when we define a macro?

After INIT_LIST_HEAD macro is called

```
     HEAD
   +-------+
.->| prev  |--.
|  +-------+  |
`--| next  |<-'
   +-------+
```

```
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...

- Sec 6.1.5, *Manipulating Linked Lists* in [Lov10]
- This function adds the new node to the given list immediately after the head node.

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

  Because the list is circular and generally has no concept of *first* or *last* nodes, you can pass any element for head. If you do pass the ”last” element, however, this function can be used to implement a stack (Sec *Manipulating Linked Lists* in [Lov10]).

List Traversing

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

list_for_each

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

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

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

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

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

list_entry(ptr, type, member)

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

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

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

- container_of(ptr,type,member): [kernelnewbies.org: Linked Lists] The code above deserves some explanation (Also read FAQ/ContainerOf). We have the address of the struct list_head field inside of another data structure (let's say struct task_struct for sake of example). The first line of the macro casts the value 0 into a pointer of the encapsulating data structure type (struct task_struct in our example). We use this pointer to access the field in that data structure which corresponds to our list_head and get its type with the macro typeof to declare a pointer mptr initialized to the value contained in ptr.

- (Linux Kernel Linked List Explained) How Does This Work?
- list_entry(ptr,type,member)

```
struct fox *f;

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

list_for_each_entry(pos, head, member)

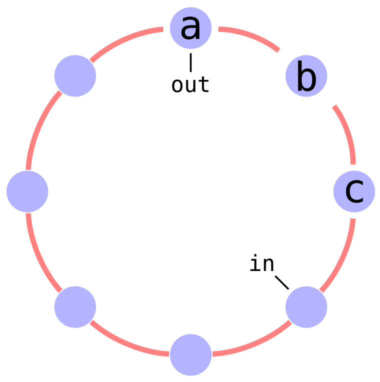
```
/**
 * list_for_each_entry  -       iterate over list of given type
 * @pos:        the type * to use as a loop counter.
 * @head:       the head for your list.
 * @member:     the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member)                      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

3.2 Queues

Textbook: Sec 6.2, *Queues*, [Lov10]

Queue (FIFO)

Circular buffer



Empty: in == out

Full: (in+1)%BUFFER_SIZE == out

Can be lock-free.

- Wikipedia: Circular buffer

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;
```

- Sec *Creating a queue*, p97, [Lov10]

kfifo operations

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

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

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

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

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

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

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

Example

1. To enqueue 32 integers into fifo

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

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

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

- kfifo_in(): (Sec *Enqueuing Data*, p98, [Lov10]) This function copies the len bytes starting at from into the queue represented by fifo.
    - On success it returns the number of bytes enqueued.
    - If less than len bytes are free in the queue, the function copies only up to the amount of available bytes.Thus the return value can be less than len or even zero, if nothing was copied.
- kfifo_out(): (Sec *Dequeuing Data*, p98, [Lov10]) This function copies at most len bytes from the queue pointed at by fifo to the buffer pointed at by to.
    - On success the function returns the number of bytes copied.
    - If less than len bytes are in the queue, the function copies less than requested.

3.3 Binary Trees

Textbook: Sec 6.4, *Binary Trees*, in [Lov10];

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:

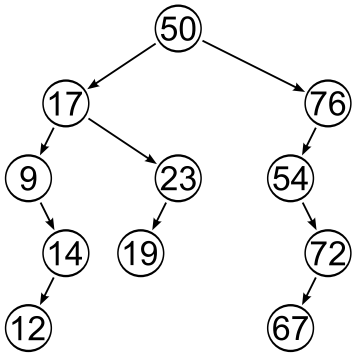
1. searching for a given node
2. in-order traversal (e.g. Left-Root-Right)

More info:

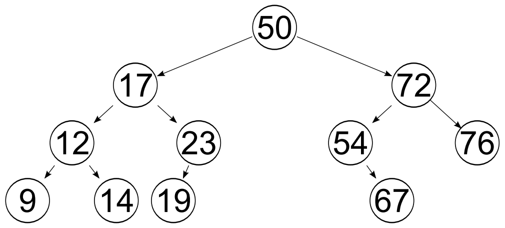
- Binary search tree
  - http://en.wikipedia.org/wiki/Binary_search_tree
  - http://www.cs.duke.edu/~reif/courses/alglectures/skiena.lectures/lecture8.pdf
  - http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/bst.pdf/
  - http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/binarySearchTree.htm

---

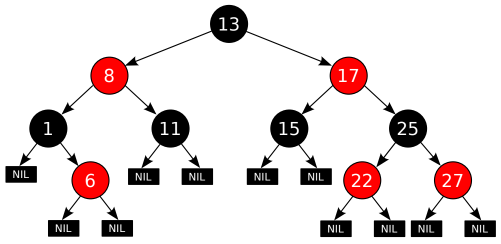
Unbalanced binary tree



Balanced binary tree



Red-black tree



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

Wikipedia articles about trees:

[Wika] Wikipedia. *AVL tree*.

[Wikb] Wikipedia. *Big O notation*.

[Wikc] Wikipedia. *Binary search tree*.

[Wikd] Wikipedia. *Binary tree*.

[Wike] Wikipedia. *Red-black tree*.

[Wikf] Wikipedia. *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*.

- Property 5 implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
- (p105, [Lov10]) Taken together, these properties ensure that the deepest leaf has a depth of no more than double that of the shallowest leaf. Consequently, the tree is always semi-balanced. Why this is true is surprisingly simple. First, by property five, a red node cannot be the child or parent of another red node. By property six, all paths through the tree to its leaves have the same number of black nodes.The longest path through the tree alternates red and black nodes.Thus the shortest path, which must have the same number of black nodes, contains only black nodes.Therefore, the longest path from the root to a leaf is no more than double the shortest path from the root to any other leaf.
- Wikipedia: Red-black tree
- Sec 8.2, Red-black tree, Data structure and algorithms
- Red-black tree Java applet demo

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

---

- lwn.net: Red-black trees
- Documentation/rbtree.txt

<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

- `http://www.kerneltravel.net/jiaoliu/kern-rbtree.html`

- (struct rb_root){ NULL, } is a compound literal. { NULL, } is an *initializer list* (Sec 6.7.8, p125, *Initialization*, C99). It initializes the 1st element (a pointer) to NULL, and initializes the rest (omitted) as if they are static objects: arithmetic types are initialized to 0; pointers are initialized to NULL(ibm: Designated initializers for aggregate types (C only)).
  - Sec 6.5.2.5, p75, Compound literals, C99
  - GCC manual: Designated Initializers

Example

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

Search

```
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;
}
```

- include/linux/rbtree.h
- p106-107, [Lov10]
- Sec 9.3, [BC05]

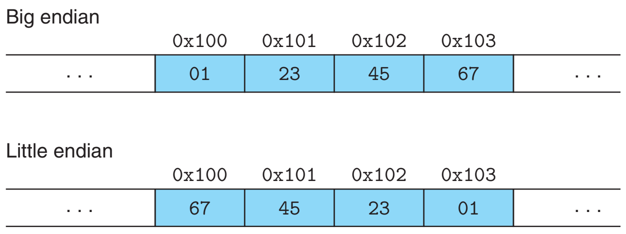
4 Assembly

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:



* Wikipedia: Endianness

```
        EAX   AX             ESI
+--------+----+----+ +--------+--------+ +---------+ +---------+
|        | AH | AL | |        |   SI   | |   CS    | |   DS    |
+--------+----+----+ +--------+--------+ +---------+ +---------+
                                                     16        0
        EBX   BX             EDI
+--------+----+----+ +--------+--------+ +---------+ +---------+
|        | BH | BL | |        |   DI   | |   ES    | |   FS    |
+--------+----+----+ +--------+--------+ +---------+ +---------+

        ECX   CX             EBP
+--------+----+----+ +--------+--------+ +---------+ +---------+
|        | CH | CL | |        |   BP   | |   GS    | |   SS    |
+--------+----+----+ +--------+--------+ +---------+ +---------+

        EDX   DX             ESP                EFLAGS
+--------+----+----+ +--------+--------+  +---------+--------+
|        | DH | DL | |        |   SP   |  |         | FLAGS  |
+--------+----+----+ +--------+--------+  +---------+--------+
31                 0 31                 0  31                 0
```

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

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

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

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

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

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

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

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

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

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

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

- stackoverflow: stos

ESI The source index, ESI, has the same properties as the destination index. The only difference is that the source index is for reading instead of writing. Although all data-processing routines write, not all read, so the source index is not as universally useful. When the time comes to use it, however, the source index is just as powerful as the destination index, and has the same type of instructions.

In situations where your code does not read any sort of data, of course, using the source index for convenient storage space is acceptable.

ESP and EBP When a block of code calls a function, it pushes the parameters and the return address on the stack. Once inside, function sets the base pointer equal to the stack pointer and then places its own internal variables on the stack. From that point on, the function refers to its parameters and variables relative to the base pointer rather than the stack pointer. Why not the stack pointer? For some reason, the stack pointer lousy addressing modes. In 16-bit mode, it cannot be a square-bracket memory offset at all. In 32-bit mode, it can be appear in square brackets only by adding an expensive SIB byte to the opcode.

In your code, there is never a reason to use the stack pointer for anything other than the stack. The base pointer, however, is up for grabs. If your routines pass parameters by register instead of by stack (they should), there is no reason to copy the stack pointer into the base pointer. The base pointer becomes a free register for whatever you need.

Segment registers

CS Code segment

SS Stack segment

DS,ES,FS,GS Data segment

An memory address is an offset in a segment

ES:EDI references memory in the ES (extra segment) with an offset of the value in the EDI

DS:ESI

CS:EIP

SS:ESP

State/Control registers

EFLAGS Status, control, and system flags

EIP The instruction pointer, contains an offset from CS (CS:EIP)

FLAGS

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

CF Carry flag

ZF Zero flag

SF Sign flag, Negative flag

OF Overflow flag

- Wikipedia — EFLAGS

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

```
     +---------+
     |         |
     +         +
     |  4th    |
     +         +
     |  word   |
     +         +
     |         |
2014 +---------+ 12
     ~         ~
     +---------+
     | 4th B   |
2005 +---------+ 3
     |         |
     +---------+ 2
     |         |
     +---------+ 1
     |         |
2002 +---------+ 0
     7         0
```

- Chapter 2, Section [Data Accessing Methods] in [Bar09]
  - in *Indexed addressing mode*, if you have a set of numbers starting at location 2002, you can cycle between each of them using an index register.
  - the *multiplier* allows you to access memory a byte at a time or a word at a time (4 bytes).

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

Example (AT&T syntax)

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

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

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

- each word is 4 bytes long
- stack grows downward
- movl — long, 32 bits
- %eax — extended, 32 bits

- 4(%esp) — base pointer addressing mode.

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

Example **(AT&T syntax)**

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

- *GAS Syntax* [Wik12].
- ch3 in [Bar09], *Addressing Modes*.
- movl — Data operation. load data into somewhere(register/memory).
- leal — Address operation. load effective address. Appendix B, p259 of [Bar09].

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
```

4.1 Assembly Language Examples

Stack Setup

Before executing a function, the program

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

ℹ

More info:

- ch4, p54 of [Bar09].

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

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

ℹ

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

ℹ

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.

- Wikipedia: Call stack: Return processing
- (p58 in [Bar09]) The calling code also needs to pop off all of the parameters it pushed onto the stack in order to get the stack pointer back where it was (you can also simply add 4 * number of paramters to %esp using the addl instruction, if you don't need the values of the parameters anymore).

Generating Assembly From C Code

simple.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
```

More info:

- info gas 'Pseudo Ops'
- Stackoverflow: What is .cfi and .LFE in assembly code produced by GCC from c++ program?
  - .LFB — Local Function Begin
  - .LFE — Local Function End
- Stackoverflow: GAS: Explanation of .cfi_def_cfa_offset
- Stackoverflow: Understanding gcc generated assembly
- Assembler directives
- cfi directives

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)

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

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

Generating Assembly From C Code

count.c

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

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

  return 0;
}
```

gcc -S count.c

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

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

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
```

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

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

- Stackoverflow:what is “stack alignment”?
- Stackoverflow: What does it mean to align the stack?
- Wikipedia: Data structure alignment

4.2 Inline Assembly

Inline Assembly

Construct

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

Example

```
asm (”movl %eax, %ebx”);
```

```
asm (”movl %eax, %ebx” :::);
```

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

```
__asm__ (”movl %eax, %ebx”);
```

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

Why inline? (Inline Functions, p227 of [Bar09]) ... reduces the function-call overhead. Functions are great from the point of view of program management - they make it easy to break up your program into independent, understandable, and reuseable parts. However, function calls do involve the overhead of pushing arguments onto the stack and doing the jumps (remember locality of reference - your code may be swapped out on disk instead of in memory). For high level languages, it’s often impossible for compilers to do optimizations across function-call boundaries. However, some languages support inline functions or function macros. These functions look, smell, taste, and act like real functions, except the compiler has the option to simply plug the code in exactly where it was called. This makes the program faster, but it also increases the size of the code. There are also many functions, like recursive functions, which cannot be inlined because they call themselves either directly or indirectly.

GCC Extended Assembly ([ibi12]) In basic inline assembly, we had only instructions. In extended assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers. *It is not mandatory to specify the registers to use, we can leave that headache to GCC and that probably fit into GCC's optimization scheme better*.

If there are a total of n operands (both input and output inclusive), then the first output operand is numbered 0, continuing in increasing order, and the last input operand is numbered n-1.

%0 the 1st output operand

%(n-1) the last input operand

Example: exit(0)

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

Quick System Call Review: (p28 in [Bar09]) To recap — Operating System features are accessed through system calls. These are invoked by setting up the registers in a special way and issuing the instruction int $0x80. Linux knows which system call we want to access by what we stored in the %eax register. Each system call has other requirements as to what needs to be stored in the other registers. System call number 1 is the exit system call, which requires the status code to be placed in %ebx.

Example

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

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

Extended asm If in our code we touch (ie, change the contents) some registers and return from asm without fixing those changes, something bad is going to happen. This is because GCC have no idea about the changes in the register contents and this leads us to trouble, especially when compiler makes some optimization. It will suppose that some register contains the value of some variable that we might have changed without informing GCC, and it continues like nothing happened. What we can do is either use those instructions having no side effects or fix things when we quit or wait for something to crash. This is where we want some extended functionality. Extended asm provides us with that functionality. (Sec 4 of [ibi12])

5 Quirky C Language Usage

Quirky C Language Usage
— *asmlinkage and fastcall*

asmlinkage tells the compiler to pass parameters on the local stack

fastcall tells the compiler to pass parameters in the general-purpose registers

Example

- asmlinkage int sys_fork(struct pt_regs regs)
- fastcall unsigned int do_IRQ(struct pt_regs *regs)

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

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

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

Why asmlinkage? (quoted from asmlinkage purposes and optimizations)

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

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

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

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

More info:

- Stackoverflow: Is "asmlinkage" required for a c function to be called from assembly?
- `http://hi.baidu.com/fiction_junru/blog/item/75ee131e94c397c3a78669d1.html`

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 */
```

ℹ

- `http://cboard.cprogramming.com/c-programming/130111-uint32_t-~0ul.html`

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

  $$\sim 0UL = 0xffffffff = 4294967295$$

  Thats the largest value that can be in a 32 bit integer.

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)
```

ℹ

More info:

- `http://www.greenend.org.uk/rjk/tech/inline.html`
- `http://en.wikipedia.org/wiki/Inline_function`

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;`

More info: `http://en.wikipedia.org/wiki/Volatile_variable`

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

ℹ

- sec 2.4 in [SBP07]
- GCC manual: Variable Attributes

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
```

ℹ

More info:

- Executable and Linkable Format
- Understanding ELF using readelf and objdump

How to decompress vmlinuz? (`http://comments.gmane.org/gmane.linux.kernel.kernelnewbies/42926`) To exam the kernel image with objdump, you have to decompress it first.

1. cp /boot/vmlinuz-3.7.0-rc3-next-20121029 /tmp/
2. od -A d -t x1 vmlinuz-3.7.0-rc3-next-20121029 | grep '1f 8b 08 00'
   The output should be something like this:

   0016992 f3 a5 fc 5e 8d 83 b4 91 4f 00 ff e0 1f 8b 08 00
3. dd if=vmlinuz bs=1 skip=17004 | zcat > vmlinux
4. How did i calculated 17004 ?

   0016992 + offset of GZ signature(1f 8b 08 00), i.e.
   0016992 + 12
5. dd if=vmlinuz-3.7.0-rc3-next-20121029 bs=1 skip=17004 | zcat > vmlinux

   ```
   5233764+0 records in
   5233764+0 records out
   5233764 bytes (5.2 MB) copied
   ```
6. file vmlinux
7. objdump -f vmlinux

8 Kernel Speak: Listen to Kernel Messages

Listening 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