Before Diving Into The Kernel Source

Wang Xiaolin

June 10, 2013

✉ wx672ster+os@gmail.com

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

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

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));
```

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)

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

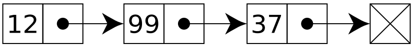
Many objects and structures in the kernel

- memory pages
- processes
- interrupts
- ...

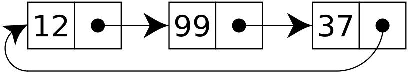
1. linked-lists — to group them together
2. binary search trees — to efficiently find a single element

Linked Lists

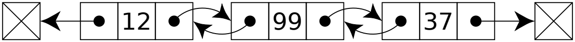
Singly linked list



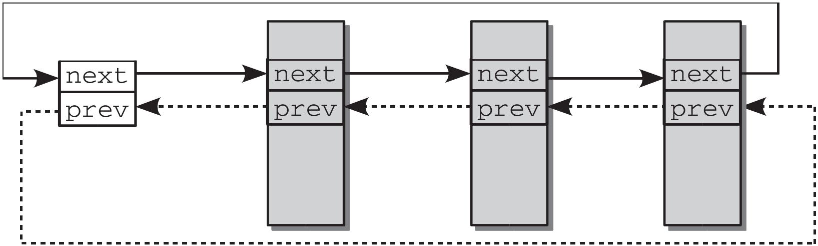
Circularly linked list



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

```c
#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

```
      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...

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.

```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?

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)
```

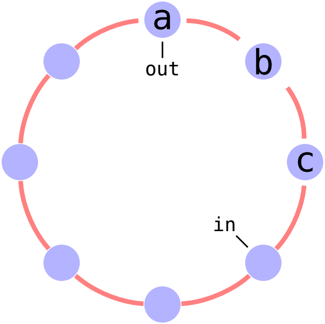
```c
struct fox *f;

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);

/* 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));
```

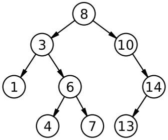
2. To dequeue and print all the items in the queue

```
/* 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);
 }
```

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



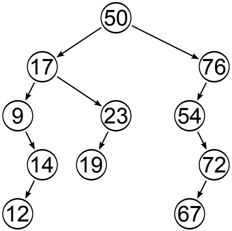
Properties:

- 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.

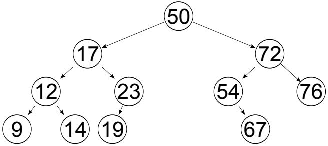
Efficient in:

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

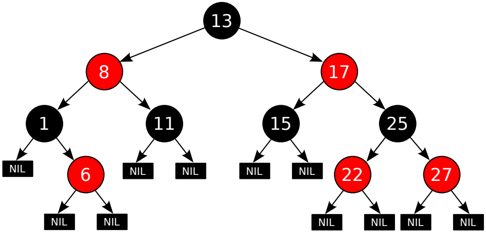
Unbalanced binary tree

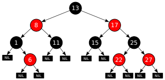


Balanced binary tree



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 $O(log n)$) 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

```c
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;
}
```

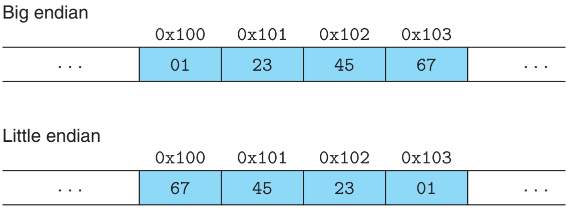
x86 Assembly

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`:



```
       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

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

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 $\boxed{\texttt{movl ADDRESS, \%eax}}$

- load data at ADDRESS into %eax

indexed addressing mode $\boxed{\texttt{movl START(,\%ecx,1), \%eax}}$

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

indirect addressing mode $\boxed{\texttt{movl (\%eax), \%ebx}}$

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

base pointer addressing mode $\boxed{\texttt{movl 4(\%eax), \%ebx}}$

immediate mode $\boxed{\texttt{movl \$12, \%eax}}$

without $ Direct addressing

indexed addressing mode

$\boxed{\texttt{movl START(,\%ecx,1), \%eax}}$

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

START(,INDEX,MULTIPLIER):

- to access the 4^{th} byte from location 2002

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

- to access the 4^{th} 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
```

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)

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

Example

```c
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.

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)))
```

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

```c
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:

```c
static volatile int foo;
```

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

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
```

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