Linux Internals
Day 1- Afternoon
Introduction to the Linux Kernel

Joel Isaacson

Ascender Technologies Ltd

The Source

- The normal way to obtain the kernel source is from the Internet as a compressed tar file.
- The current size of this file is:
  - 42 Megabytes compressed
  - 250 Megabytes uncompressed.

Directory Structure

- The “arch” subdirectory contains all the support for different machine architectures.
- This is where non-portable code should be located.
- Many architectures are supported

Supported architectures

- alpha
- arm
- arm26
- cris
- frv
- h8300
- i386
- ia64
- m32r
- m68k
- m68knommu
- mips
- parisc
- powerpc
- ppc
- s390
- sh
- sh64
- sparc
- sparc64
- um
- v850
- x86_64
- xtensa

Include files

- The include files also have machine dependent directories.
  - asm-i386
  - asm-ia64
- The “asm” directory is a symbolic link to a particular architecture directory (e.g. asm-i386)

Portable Source

- All other sources are portable code.
- The vast majority of code is portable.
- There a other notable directories
  - kernel
  - drivers
  - mm
  - net
  - fs
  - init
  - sound

A Stroll Down A Link

Linked Lists

- There is nothing simpler than a linked list.
- We will examine the linked list implementation in Linux and we will see that even simplicity can be deceptive.
- This example will give us a gentle introduction to the style and structure of Linux.

Linux Link Implementation

- The kernel has an interesting implementation of a linked list that is a good example of the organization and the coding style of the Linux kernel.
- This example also illustrates some of the design rules Linux coding.
- Of course everything C code – more or less.

The `list_head` Structure

```
// From include/linux/list.h

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

Use of list_head

- This is pretty useless as is.
- Any structure that we want to link together as a linked list we just add list_head as an element:

```
struct mylist{
   int a;
   struct list_head list;
int b;
} ml;
```

Initializing the List

- First we initialize the list.

```
INIT_LIST(&ml.list);

// from include/linux/list.h
static inline void
INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;
}
```

Adding to List

- We can add a link to the list.

```
static inline
void list_add(struct list_head *new, struct list_head *head)
{
        __list_add(new, head, head->next);
}

static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}
```

Check if list is empty?

```
static inline int list_empty(const struct list_head *head)
{
        return head->next == head;
}
```

Traversing the List

```
struct list_head *p;

list_for_each(p, &lm.list){
    struct my_struct *m;
    m= list_entry(p, struct my_struct, list);
}
```

Now things are getting weird.
Have a look at the last two arguments
to “list_entry”

Traversing the List

```
#define list_for_each(pos, head) \
for (pos = (head)->next; prefetch(pos->next), pos != (head); \
             pos = pos->next)
```

The prefetch() function will do a speculative load of the next element. It is defined as a null operator in some architectures.

list_entry

```
#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) );})

#ifdef __compiler_offsetof
#define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
#else
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#endif
```

Not very nice code.
Linus tends to put all the ugly things in 'h' files.
The C code is readable.

Processes

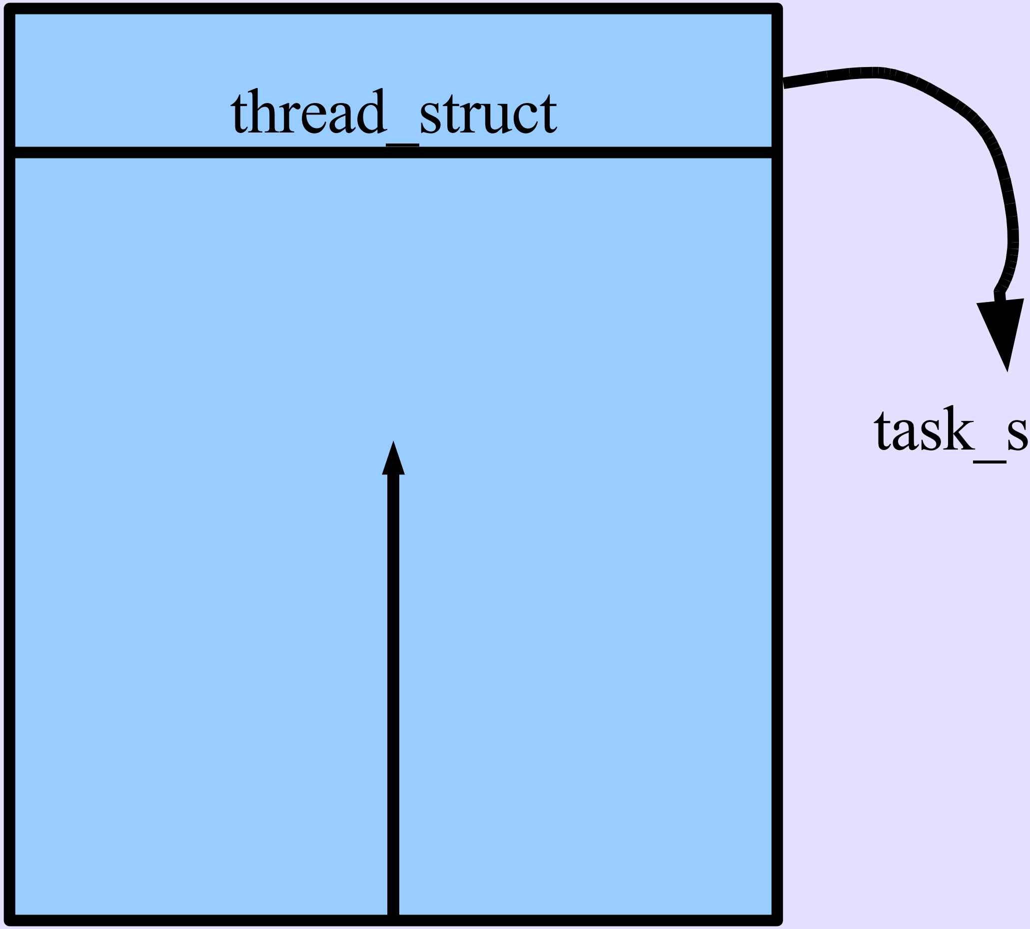
Process == Task == Thread

Linux uses these three names interchangeably

The Process Table

- The process table consists of a collection of objects of types “struct task_struct”.
- Each process has a “struct thread_struct” at the end of the kernel process stack that has as its first element a pointer to “task_struct”.
- The process table can be enumerated by a link in the “task_struct”

Kernel Mode Stack



task_struct

4 or 8 Kbytes
page aligned

Time for a quiz

```
{
   xtype *p;
   char *q;
   p= (xtype *) *(long *)(((long) q) & ~0x1fff);
}
```

The question is:
What does “p” point to?
What type is “xtype”?
Hint: the kernel stack is 8K bytes.

The Answer is ...

- This code is very strange.
- I normally wouldn't show this code but it is used heavily in the kernel via inline routines.
- Even experienced kernel programmer might not recognize it since it usually is hidden deep within the processor dependent include files.

The Answer

- This code is used to return the “task_struct” of the user process.
- The tricky idea is that any address on the kernel mode stack when aligned to the nearest 8 Kb boundary will point to the “thread_struct”.
- The first element of the “thread_struct” points to the “task_struct” which is the process entry.

current_thread_info()
asm/thread_info.h

```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp")
                      __attribute_used__;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)(current_stack_pointer
              & ~(THREAD_SIZE - 1));
}
```

current
asm/current.h

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

#define current get_current()
```