Linux Architecture-Specific Kernel Interfaces

Michael Hohmuth

March 21, 1996

Contents

Introduction

This document corresponds to Linux 1.3.78++.

Chapter 1

include/asm-* Interface

Architectures can implement these interfaces in any way they want (macros, inline functions, inline asm), but preferably fast. :-)

These include files also contain architecture-specific type definitions and system call parameters—see section 1.22.

1.1 <asm/atomic.h>

Atomic operations that C can't guarantee us. Useful for resource counting etc..

```
typedef int atomic_t;

static __inline__ void atomic_add(atomic_t i, atomic_t *v);
static __inline__ void atomic_sub(atomic_t i, atomic_t *v);
static __inline__ void atomic_inc(atomic_t *v);
static __inline__ void atomic_dec(atomic_t *v);
static __inline__ int atomic_dec_and_test(atomic_t *v);
```

1.2 <asm/bitops.h>

Defines:

```
int set_bit(int nr, void * addr);
int clear_bit(int nr, void * addr);
int change_bit(int nr, void * addr);
```

Atomic bit operations (return old bit value).

```
int test_bit(int nr, void * addr);
int find_first_zero_bit(void * addr, unsigned size);
int find_next_zero_bit (void * addr, int size, int offset);
```

More bit operations (don't need to be atomic).

```
unsigned long ffz(unsigned long word);
```

Find First Zero in word. Undefined if no zero exists, so code should check against ~0UL first.

1.3 <asm/byteorder.h>

Device driver support:

```
#define __LITTLE_ENDIAN 1234
#define __BIG_ENDIAN
```

This include file needs to specify exactly one of the above two definitions...

```
#define __LITTLE_ENDIAN_BITFIELD
#define __BIG_ENDIAN_BITFIELD
```

...and exactly one of these two definitions.

```
#define ntohl(x) ...
#define ntohs(x) ...
#define htonl(x) ...
#define htons(x) ...
```

Convert longs and shorts from host to network byte order, and vice versa.

1.4 <asm/bugs.h>

This is included by `init/main.c` to check for architecture-dependent bugs.

Defines:

```
static void check_bugs(void);
```

Called at kernel initialization; should check for processor bugs and set global variables associated with them.

1.5 <asm/checksum.h>

Compute several TCP/IP checksums. For Linux/L4, can probably be copied from `<asm-i386/dma.h>`.

1.6 <asm/delay.h>

```
extern __inline__ void __delay(int loops);
```

Loop for `loops` loops.

```
extern __inline__ void udelay(unsigned long usecs);
```

Loop for `usecs` micro seconds using the global variable `unsigned long loops_per_sec` (calibrated at kernel initialization).

```
extern __inline__ unsigned long muldiv(unsigned long a,
        unsigned long b, unsigned long c);
```

Compute `a*b/c`.

1.7 <asm/dma.h>

DMA controller manipulation via ports. For Linux/L4, can probably be copied from `<asm-i386/dma.h>`.

1.8 <asm/floppy.h>

i386-specific floppy device driver support. For Linux/L4, can probably be copied from `<asm-i386/floppy.h>`.

1.9 <asm/io.h>

Port manipulation (`inb()`, `outb()` and the like). For Linux/L4, can probably be copied from `<asm-i386/io.h>`.

1.10 <asm/irq.h>

Defines:

```
void disable_irq(unsigned int);
void enable_irq(unsigned int);
```

Mask/unmask the specified IRQ.

1.11 <asm/mmu_context.h>

```
inline void get_mmu_context(struct task_struct *p);
```

Get a new MMU context (ASN, address space number). (The x86's don't know about contexts.)

1.12 <asm/page.h>

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#define PAGE_MASK       (~ (PAGE_SIZE-1))

typedef unsigned long pte_t;
typedef unsigned long pmd_t;
typedef unsigned long pgd_t;
typedef unsigned long pgprot_t;
```

Typedefs for hardware data structures: page table entry, page mid-level directory entry, page directory entry, page protection flags.

```
#define pte_val(x)      (x)
#define pmd_val(x)      (x)
#define pgd_val(x)      (x)
#define pgprot_val(x)   (x)
```

Access functions for the above types. (This allows to implement type checks.)

```
#define __pte(x)        (x)
#define __pgd(x)        (x)
#define __pgprot(x)     (x)
```

These macros cast a value to a given type.

```
#define PAGE_ALIGN(addr)        (((addr)+PAGE_SIZE-1)&PAGE_MASK)
```

To align the pointer to the (next) page boundary.

```
#define PAGE_OFFSET             0
#define MAP_NR(addr)            (((unsigned long)(addr)) >> PAGE_SHIFT)
```

This handles subscripting the memory map array mem_map[] (defined in <linux/mm.h>).

1.13 <asm/param.h>

```
#ifndef HZ
#define HZ 100
#endif

#define EXEC_PAGESIZE   4096

#ifndef NGROUPS
#define NGROUPS         32
#endif

#ifndef NOGROUP
#define NOGROUP         (-1)
#endif

#define MAXHOSTNAMELEN  64      /* max length of hostname */
```

The obvious stuff.

1.14 <asm/pgtable.h>

```
static inline void invalidate(void);
static inline void invalidate_all(void);
static inline void invalidate_mm(struct mm_struct *mm);
static inline void invalidate_page(struct vm_area_struct *vma,
        unsigned long addr);
static inline void invalidate_range(struct mm_struct *mm,
        unsigned long start, unsigned long end);
```

Flush TLB entries:

- invalidate() invalidates the current mm struct TLBs
- invalidate_all() invalidates all processes TLBs

- invalidate_mm(mm) invalidates the specified mm context TLB's
- invalidate_page(mm, vmaddr) invalidates one page
- invalidate_range(mm, start, end) invalidates a range of pages

```
/* PMD_SHIFT determines the size of the area a second-level page table can map */
#define PMD_SHIFT       22
#define PMD_SIZE        (1UL << PMD_SHIFT)
#define PMD_MASK        (~ (PMD_SIZE-1))

/* PGDIR_SHIFT determines what a third-level page table entry can map */
#define PGDIR_SHIFT     22
#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
#define PGDIR_MASK      (~ (PGDIR_SIZE-1))

/*
 * entries per page directory level: the i386 is two-level, so
 * we don't really have any PMD directory physically.
 */
#define PTRS_PER_PTE    1024
#define PTRS_PER_PMD    1
#define PTRS_PER_PGD    1024
```

The above definitions define the sizes of page directories, mid-level page directories and page tables.

```
#define VMALLOC_OFFSET  (8*1024*1024)
#define VMALLOC_START ((high_memory + VMALLOC_OFFSET) & ~ (VMALLOC_OFFSET-1))
#define VMALLOC_VMADDR(x) (TASK_SIZE + (unsigned long)(x))
```

VMALLOC_START is the start of the kernel's virtual vmalloc() area—in this case, at the next 8MB boundary after the end of physical memory. The VMALLOC_VMADDR() macro computes the virtual (linear) address of a vmalloc()'d region (TASK_SIZE points to the beginning of the kernel's data segment (0xC0000000 in the monolithic kernel)).

```
#define PAGE_NONE       ... /* page mapped, but none of r/w/x */
#define PAGE_SHARED     ... /* r/w shared page */
#define PAGE_COPY       ... /* r/o mapped copy-on-write shared page */
#define PAGE_READONLY   ... /* r/o page */
#define PAGE_KERNEL     ... /* kernel-only page */
```

Different page attributes. These must be mapped to bit vectors that can be stored in page table entries.

```
        /* xwr */
#define __P000  PAGE_NONE
#define __P001  PAGE_READONLY
#define __P010  PAGE_COPY
#define __P011  PAGE_COPY
#define __P100  PAGE_READONLY
#define __P101  PAGE_READONLY
#define __P110  PAGE_COPY
#define __P111  PAGE_COPY

#define __S000  PAGE_NONE
#define __S001  PAGE_READONLY
```

```
#define __S010  PAGE_SHARED
#define __S011  PAGE_SHARED
#define __S100  PAGE_READONLY
#define __S101  PAGE_READONLY
#define __S110  PAGE_SHARED
#define __S111  PAGE_SHARED
```

Page protections (r/w/x) for private and shared cases mapped to page attributes. The i386 can't do page protection for execute, and considers them the same as read. Also, write permissions imply read permissions. This is the closest we can get.

```
#define BAD_PAGETABLE ...
#define BAD_PAGE ...
#define ZERO_PAGE ...
```

`BAD_PAGETABLE` is used when we need a bogus page-table (a table pointing to bogus pages), while BAD_PAGE is used for a bogus page (a shared scratch page). These are used when Linux is out of memory.

`ZERO_PAGE` is a global shared page that is always zero: used for zero-mapped memory areas etc.

```
/* number of bits that fit into a memory pointer */
#define BITS_PER_PTR                    (8*sizeof(unsigned long))

/* to align the pointer to a pointer address */
#define PTR_MASK                        (~ (sizeof(void*)-1))

/* sizeof(void*)==1<<SIZEOF_PTR_LOG2 */
/* 64-bit machines, beware!  SRB. */
#define SIZEOF_PTR_LOG2                 2
```

Used for pointer arithmetics.

```
#define PAGE_PTR(address) ...
```

This macro computes an index into a page table.

```
inline void SET_PAGE_DIR(struct task_struct * tsk, pgd_t * pgdir);
```

Set the task's page directory to `pgdir`. If the task is the current task, set the current hardware page directory pointer to `pgdir`.

```
extern unsigned long high_memory;
```

End of physical memory.

```
extern inline int pte_none(pte_t pte);
extern inline int pte_present(pte_t pte);
extern inline void pte_clear(pte_t *ptep);
```

Page table entry manipulation: check whether a page table entry is in use, or the page is present, and a routine to clear a page table entry.

```
extern inline int pmd_none(pmd_t pmd);
extern inline int pmd_bad(pmd_t pmd);
extern inline int pmd_present(pmd_t pmd);
extern inline void pmd_clear(pmd_t * pmdp);
```

The same routines for mid-level page directories, with an additional routine that checks whether a page directory entry is invalid.

```
extern inline int pgd_none(pgd_t pgd);
extern inline int pgd_bad(pgd_t pgd);
extern inline int pgd_present(pgd_t pgd);
extern inline void pgd_clear(pgd_t * pgdp);
```

The same routines for (3rd-level) page directories.

```
extern inline int pte_read(pte_t pte);
extern inline int pte_write(pte_t pte);
extern inline int pte_exec(pte_t pte);
extern inline int pte_dirty(pte_t pte);
extern inline int pte_young(pte_t pte);
```

These routines check whether some specific flag is set on a page. They only need to return defined results if the page table entry is valid (i.e., the page is present). "Young" means the page has been accessed.

```
extern inline pte_t pte_wrprotect(pte_t pte);
extern inline pte_t pte_rdprotect(pte_t pte);
extern inline pte_t pte_exprotect(pte_t pte);
extern inline pte_t pte_mkclean(pte_t pte);
extern inline pte_t pte_mkold(pte_t pte);
extern inline pte_t pte_uncow(pte_t pte);
extern inline pte_t pte_mkwrite(pte_t pte);
extern inline pte_t pte_mkread(pte_t pte);
extern inline pte_t pte_mkexec(pte_t pte);
extern inline pte_t pte_mkdirty(pte_t pte);
extern inline pte_t pte_mkyoung(pte_t pte);
extern inline pte_t pte_modify(pte_t pte, pgprot_t newprot);
```

These routines manipulate page table entries: They take a valid page table entry and return the modified entry. Pages can be write protected, read protected, protected from being executed, made "clean" and "dirty" (i.e., the dirty flag is cleared/set), made "old" and "young" (i.e., have their accessed bit cleared/set), made writable, readable and executable, and have their protection flags reset to some value.

```
extern inline pte_t mk_pte(unsigned long page, pgprot_t pgprot);
extern inline unsigned long pte_page(pte_t pte);
extern inline unsigned long pmd_page(pmd_t pmd);
```

Conversion functions: convert a page and protection to a page entry, and a page entry and page directory to the page they refer to.

```
extern inline pgd_t * pgd_offset(struct task_struct * tsk, unsigned
                                 long address);
extern inline pmd_t * pmd_offset(pgd_t * dir, unsigned long address);
extern inline pte_t * pte_offset(pmd_t * dir, unsigned long address);
```

These routines return entries belonging to specific virtual addresses from page directories and page tables.

```
extern inline void pte_free_kernel(pte_t * pte);
extern inline pte_t * pte_alloc_kernel(pmd_t * pmd, unsigned long address);
```

```
extern inline void pmd_free_kernel(pmd_t * pmd);
extern inline pmd_t * pmd_alloc_kernel(pgd_t * pgd, unsigned long address);
extern inline void pte_free(pte_t * pte);
extern inline pte_t * pte_alloc(pmd_t * pmd, unsigned long address);
extern inline void pmd_free(pmd_t * pmd);
extern inline pmd_t * pmd_alloc(pgd_t * pgd, unsigned long address);
extern inline void pgd_free(pgd_t * pgd);
extern inline pgd_t * pgd_alloc(void);
```

Allocate and free page tables, and return a pointer an entry in the new table that belongs to `address`. The `xxx_kernel()` versions are used to allocate a kernel page table—this turns on ASN (address space number) bits if any, and marks the page tables `MAP_PAGE_RESERVED` in `mem_map[]`.

```
extern pgd_t swapper_pg_dir[1024];
```

A kernel page directory used whenever the current task's page directory is invalid.

```
extern inline void update_mmu_cache(struct vm_area_struct * vma,
        unsigned long address, pte_t pte);
```

Update the MMU after manupulating the page tables. On the i386 hardware, this is unnecessary because it doesn't have any external MMU info.

```
#define SWP_TYPE(entry) (((entry) >> 1) & 0x7f)
#define SWP_OFFSET(entry) ((entry) >> 8)
#define SWP_ENTRY(type,offset) (((type) << 1) | ((offset) << 8))
```

Extract information about a swapped page from a page table entry, and construct a page table entry from swap info.

1.15 <asm/processor.h>

```
#define TASK_SIZE ...
```

Size of user virtual memory area. On the i386, this should be 0xC0000000.

```
extern int EISA_bus;
extern int MCA_bus;
```

Bus types.

```
extern int wp_works_ok;
```

Should be set to 0 if supervisor can access write-protected pages.

This include file should also declare all other system setup/ hardware bug flags tested for in `<asm/bugs.h>`.

```
#define EISA_bus__is_a_macro
#define MCA_bus__is_a_macro
#define wp_works_ok__is_a_macro
```

If any of the respective variables has been hard-coded using a macros, the corresponding definition should be included to avoid problems when building the kernel's symbol table (`kernel/ksyms.c`).

```
struct thread_struct {...};
```

State associated with a thread, i.e., register values, etc.

```
#define INIT_TSS {...}
```

A `thread_struct` used as an initializer for the kernel's first task.

```
#define INIT_MMAP {...}
```

A `vm_area_struct` used as an initializer for `init_mmap`, which seems to be used nowhere (go to `linux/sched.c` and see for yourself...).

```
#define alloc_kernel_stack()    ...
#define free_kernel_stack(page) ...
```

Allocate/free a page used as a kernel stack for a task.

```
static inline void start_thread(struct pt_regs * regs,
        unsigned long eip, unsigned long esp);
```

Start a newly created thread: setup `regs` using `eip` and `esp`. Also setup segment registers in `regs` if the architecture is segmented.

```
extern inline unsigned long thread_saved_pc(struct thread_struct *t);
```

Return saved PC of a blocked thread. This routine can assume the thread has been blocked by a call to `schedule()`, and can look up the PC using the thread's stack and the SP in `t`.

1.16 <asm/ptrace.h>

```
struct pt_regs {...};
```

This struct defines the way the registers are stored on the stack during a system call or exception, i.e., system call handler fill find this structure on the stack. That means that this structure needs to be layed out such that system call handlers also find their parameters at the places they expect them (at the bottom of the stack).

```
#define user_mode(regs) ...
```

Returns true if the process is currently in user mode (and not in system mode).

```
#define instruction_pointer(regs) ...
```

Return the PC as an lvalue.

```
void show_regs(struct pt_regs * regs);
```

`printk()` processor state.

1.17 <asm/segment.h>

This file defines an interface for accessing user memory (copyin/copyout).

```
static inline unsigned char get_user_byte(const char * addr);
#define get_fs_byte(addr) get_user_byte((char *)(addr))

static inline unsigned short get_user_word(const short *addr);
#define get_fs_word(addr) get_user_word((short *)(addr))

static inline unsigned long get_user_long(const int *addr);
#define get_fs_long(addr) get_user_long((int *)(addr))

static inline void put_user_byte(char val,char *addr);
#define put_fs_byte(x,addr) put_user_byte((x),(char *)(addr))

static inline void put_user_word(short val,short * addr);
#define put_fs_word(x,addr) put_user_word((x),(short *)(addr))

static inline void put_user_long(unsigned long val,int * addr);
#define put_fs_long(x,addr) put_user_long((x),(int *)(addr))

static inline void memcpy_tofs(void * to, const void * from, unsigned long n);

static inline void memcpy_fromfs(void * to, const void * from, unsigned long n);
```

Copyin/copyout routines.

```
#define KERNEL_DS ...
#define USER_DS ...
```

Descriptors for kernel and user data segments.

```
static inline unsigned long get_fs(void);
static inline unsigned long get_ds(void);
static inline void set_fs(unsigned long val);
```

The `get_` routines return the user/kernel data segment descriptors. The `set_fs()` can be used to temporarily map the user space to kernel space (so that device drivers using copyin/copyout can operate on kernel buffers); `set_fs()` only needs to accept `KERNEL_DS` and `USER_DS`.

1.18 <asm/smp.h>, <asm/smp_lock.h>

XXX

Deosn't need to define anything if `__SMP__` is not `#defined`, but could this perhaps be used to do the necessary locking when using external scheduling and kernel-internal multi-threading?

1.19 <asm/string.h>

```
extern inline void * memcpy(void * to, const void * from, size_t n);
extern inline void * memset(void * s, char c, size_t count);
extern inline size_t strlen(const char * s);
```

These are the minimally required routines. `<asm-i386/string.h>` defines many more; this file can probably copied for Linux/L4.

1.20 <asm/system.h>

```
#define switch_to(prev,next) ...
```

Switch the context to task `next`, making the task "current" (`current = next`). This also switches the user address space to `next`'s address space.

```
#define nop() ...
#define sti() ...
#define cli() ...
#define iret() ...
#define mb() ...
#define xchg(ptr,x) ...

extern inline unsigned long xchg_u8(char * m, unsigned long val);
extern inline unsigned long xchg_u16(short * m, unsigned long val);
extern inline unsigned long xchg_u32(int * m, unsigned long val);
extern inline unsigned long xchg_u64(volatile long * m, unsigned long val);
```

Some machine instructions available to i386 device drivers. `mb()` is a memory barrier instruction; it prevents the optimizer from moving memory references across it.

```
extern inline int tas(char * m);
```

Atomic test-and-set instruction.

```
#define save_flags(x)
#define restore_flags(x)
```

Save processor flags in a longword.

```
#define HAVE_DISABLE_HLT
void disable_hlt(void);
void enable_hlt(void);
```

A non-mandantory interface available to device drivers.

1.21 <asm/unistd.h>

This include file `#defines` numbers for all system calls.

In addition, it defines:

XXX

1.22 ABI Definitions

Several include files merely define constants and structures pertaining to the architecture's ABI (application binary interface).For Linux/L4, can probably be copied from `<asm-i386/*>`.

1.22.1 <asm/a.out.h>

Defines:

```
struct exec {...};
```

a.out exec structure for this architecture.

```
#define N_TRSIZE(a)     ((a).a_trsize)
#define N_DRSIZE(a)     ((a).a_drsize)
#define N_SYMSIZE(a)    ((a).a_syms)
```

Macros for accessing `struct execs`.

1.22.2 <asm/elf.h>

```
#define ELF_NGREG ...
typedef ... elf_greg_t;
typedef elf_greg_t elf_gregset_t[ELF_NGREG];
```

ELF general-purpose register and register set types.

```
typedef ... elf_fpregset_t;
```

ELF floating-point register set type.

1.22.3 <asm/errno.h>

Error numbers.

1.22.4 <asm/fcntl.h>

Several `fcntl()` opcodes.

1.22.5 <asm/ioctl.h>, <asm/ioctls.h>

Several `ioctl()` macros.

1.22.6 <asm/mman.h>

Several `mmap()` opcodes.

1.22.7 <asm/posix_types.h>

Several POSIX type definitions.

1.22.8 <asm/resource.h>

Resource limits for this architecture.

1.22.9 <asm/shmparam.h>

Shared memory definitions.

1.22.10 <asm/signal.h>

Signal handler types, signal numbers, `struct sigaction`, `sigaction()` options.

1.22.11 <asm/sigcontext.h>

```
struct sigcontext {...};
```

Signal context structure, passed as the third argument to signal handlers.

1.22.12 <asm/socket.h>, <asm/sockios.h>

Socket `setsockoptions()` and `ioctl()` opcodes.

1.22.13 <asm/stat.h>

```
struct old_stat {...};
struct new_stat {...};
```

XXX

1.22.14 <asm/statfs.h>

```
struct statfs {...};
```

1.22.15 <asm/termios.h>, <asm/termbits.h>

`struct termios`, `struct termio`, and friends.

1.22.16 <asm/types.h>

Several type definitions.

1.22.17 <asm/user.h>

Core file format.

Chapter 2

Architecture-Specific `include/linux` Interface

Architectures must implement these interfaces exactly as defined in the `include/linux` header files.

2.1 <linux/bios32.h>

XXX

2.2 <linux/interrupt.h>

```
extern unsigned int probe_irq_on(void);
extern int probe_irq_off(unsigned int);
```

Autoprobing for irqs: `probe_irq_on()` and `probe_irq_off()` provide robust primitives for accurate IRQ probing during kernel initialization. They are reasonably simple to use, are not "fooled" by spurious interrupts, and, unlike other attempts at IRQ probing, they do not get hung on stuck interrupts (such as unused PS2 mouse interfaces on ASUS boards). For reasonably foolproof probing, use them as follows:

1. clear and/or mask the device's internal interrupt.
2. `sti();`
3. `irqs = probe_irq_on(); // ``take over'' all unassigned idle IRQs`
4. enable the device and cause it to trigger an interrupt.
5. wait for the device to interrupt, using non-intrusive polling or a delay.
6. `irq = probe_irq_off(irqs); // get IRQ number, 0=none, negative=multiple`
7. service the device to clear its pending interrupt.
8. loop again if paranoia is required.

`probe_irq_on()` returns a mask of snarfed irq's, or 0 on failure.

`probe_irq_off()` takes the mask as a parameter, and returns the irq number which occurred, or zero if none occurred, or a negative irq number if more than one irq occurred.

2.3 <linux/mm.h>

```
void show_mem(void);
```

`printk()` information about memory usage.

```
void si_meminfo(struct sysinfo *val);
```

Dump memory information to `val`.

2.4 <linux/sched.h>

```
int request_irq(unsigned int irq,
        void (*handler)(int, void *, struct pt_regs *),
        unsigned long flags, const char *devname, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

These routines register/unregister interrupt handlers.

irq is the IRQ number.

handler is the interrupt handler. It receives the IRQ number, the `dev_id` (see below) and a `struct pt_regs` pointer as arguments.

On the i386, passing the registers is pretty pointless: I didn't find a single driver that makes any sensible use of it. (Well, the keyboard driver can print the register values when pressing some hotkey, but I did say *sensible* use, didn't I?)

flags The `request_irq()` routine must understand the following flags (defined in `<asm-*/signal.h>`):

- **SA_INTERRUPT** Install a "fast" interrupt handler; fast handlers run with interrupts disabled, and they don't receive a valid `pt_regs` value.

  Otherwise, the handler will be "slow:" Slow handlers run with interrupts enabled. When they return, the kernel should execute bottom halfs, deliver signals and reschedule if necessary.

- **SA_SHIRQ** Allow this interrupt to be shared with other handlers.

- **SA_SAMPLE_RANDOM** This interrupt happens unpredictably enough be suitable for random sampling. This is used to help the kernel's random number generator (see also section Interrupts (4.2.4)). `request_irq()` should call `rand_initialize_irq(irq)` (declared in `<linux/random.h>`).

devname A human-readable device name associated with this interrupt.

dev_id A unique ID associated with this handler, used as a handle to de-install the handler.

(See the Kernel Hacker's Guide for a more verbose description.)

```
void copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
        struct task_struct * p, struct pt_regs * regs);
void exit_thread(void);
void flush_thread(void);
void dump_thread(struct pt_regs * regs, struct user * dump);
void release_thread(struct task_struct *dead_task);
```

Copy the current thread with registers `regs` to `p`, exit a thread, recycle a thread (i.e., free a thread's resources), release a thread (i.e., free a dead thread that already had been `exit_thread()`'d) and dump a thread's state into a user strucure, respectively.

2.5 <linux/time.h>

```
void do_gettimeofday(struct timeval *tv);
void do_settimeofday(struct timeval *tv);
```

Get/set time using the machine's hardware.

2.6 <linux/vm86.h>

XXX

Chapter 3

Other Architecture-Specific Interfaces

3.1 System Calls

```
asmlinkage int sys_ioperm(unsigned long from, unsigned long num, int turn_on);
asmlinkage int sys_iopl(long ebx,long ecx,long edx,
        long esi, long edi, long ebp, long eax, long ds,
        long es, long fs, long gs, long orig_eax,
        long eip,long cs,long eflags,long esp,long ss);
asmlinkage int sys_modify_ldt(int func, void *ptr, unsigned long bytecount);
asmlinkage int sys_fork(struct pt_regs regs);
asmlinkage int sys_clone(struct pt_regs regs);
asmlinkage int sys_execvesys_execve(struct pt_regs regs);
asmlinkage int sys_idle(void);
asmlinkage int sys_ptrace(long request, long pid, long addr, long data);
asmlinkage int sys_sigsuspend(int restart, unsigned long oldmask,
        unsigned long set);
asmlinkage int sys_sigreturn(unsigned long __unused);
asmlinkage int sys_pipe(unsigned long * fildes);
asmlinkage int sys_vm86(struct vm86_struct * v86);
```

The i386 support must implement these system calls. See the Linux man pages for a description.

XXX

3.2 `proc` Filesystem Support

```
int get_irq_list(char *buf);
```

Print a description of IRQs to `buf`.

```
int get_cpuinfo(char *buf);
```

Print CPU information to `buf`.

```
#ifdef __SMP_PROF__
int get_smp_prof_list(char *buf);
#endif
```

Print SMP profiling information to `buf`.

3.3 init/main.c Support

```
void setup_arch(char **cmdline_p, unsigned long *memory_start_p,
        unsigned long *memory_end_p);
```

setup_arch (linuxarch/i386/kernel/setup.c/)initializes some global structures containing information about the configuration of the machine linux is running on. It expects the information at a well defined address (empty_zero_page), where it is put by code sequences in `setup.S`.

If we remove all unnessecary things, setup_arch initializes at least the following structures:

ROOT_DEV Setup the root device. It is taken from `ORIG_ROOT_DEV (*(unsigned short *) (PARAM+0x1FC))`, but I don't know were it really comes from. (There is a label in bootsect.S, but I don't know, how the value is transfered to empty_zero_page. It seems, that setup.S moves the values to 0x9000 and head.S moves them to empty_zero_page)

drive_info Drive info is a 32 byte structure containing information about hard disks. It is taken from `DRIVE_INFO (*(struct drive_info_struct *) (PARAM+0x80))`

screen_info `SCREEN_INFO (*(struct screen_info *) (PARAM+0))`

```
/*
 * These are set up by the setup-routine at boot-time:
 */

struct screen_info {
unsigned char  orig_x;
unsigned char  orig_y;
unsigned char  unused1[2];
unsigned short orig_video_page;
unsigned char  orig_video_mode;
unsigned char  orig_video_cols;
unsigned short unused2;
unsigned short orig_video_ega_bx;
unsigned short unused3;
unsigned char  orig_video_lines;
unsigned char  orig_video_isVGA;
unsigned short orig_video_points;
};
```

auxillary device info `AUX_DEVICE_INFO (*(unsigned char *) (PARAM+0x1FF))`

set by setup.S, ps_mouse detected or not

memory_end and meory_start

init_task.mm

other

```
        ROOT_DEV = to_kdev_t(ORIG_ROOT_DEV);
        drive_info = DRIVE_INFO;
        screen_info = SCREEN_INFO;
#ifdef CONFIG_APM
```

```
        apm_bios_info = APM_BIOS_INFO;
#endif
        aux_device_present = AUX_DEVICE_INFO;
        memory_end = (1<<20) + (EXT_MEM_K<<10);
        memory_end &= PAGE_MASK;
#ifdef CONFIG_BLK_DEV_RAM
        rd_image_start = RAMDISK_FLAGS & RAMDISK_IMAGE_START_MASK;
        rd_prompt = ((RAMDISK_FLAGS & RAMDISK_PROMPT_FLAG) != 0);
        rd_doload = ((RAMDISK_FLAGS & RAMDISK_LOAD_FLAG) != 0);
#endif
#ifdef CONFIG_MAX_16M
        if (memory_end > 16*1024*1024)
                memory_end = 16*1024*1024;
#endif
        if (!MOUNT_ROOT_RDONLY)
                root_mountflags &= ~ MS_RDONLY;
        memory_start = (unsigned long) &_end;
        init_task.mm->start_code = TASK_SIZE;
        init_task.mm->end_code = TASK_SIZE + (unsigned long) &_etext;
        init_task.mm->end_data = TASK_SIZE + (unsigned long) &_edata;
        init_task.mm->brk = TASK_SIZE + (unsigned long) &_end;

        /* Save unparsed command line copy for /proc/cmdline */
        memcpy(saved_command_line, COMMAND_LINE, COMMAND_LINE_SIZE);
        saved_command_line[COMMAND_LINE_SIZE-1] = '\0';
```

```
void init_IRQ(void);
void trap_init(void);
void arch_syms_export(void);
```

XXX

```
unsigned long paging_init(unsigned long start_mem, unsigned long end_mem);
```

Set up the page tables: map real memory into the kernel's user space, fix up `start_mem` accordingly. Also unmap the page at virtual kernel address 0, so that we can trap those pesky NULL-reference errors in the kernel. Then call `free_area_init(start_mem, end_mem);` (not declared anywhere).

```
void mem_init(unsigned long start_mem, unsigned long end_mem);
```

Setup the `mem_map[]` array. For each available page, call `free_page()` (declared in `<linux/mm.h>`).

3.4 Misc. Other Stuff

```
int dump_fpu (struct user_i387_struct* fpu);
```

Dump FPU state.

```
void hard_reset_now(void);
```

Reset the CPU and reboot.

Chapter 4

Kernel Interface to Architecture Support

This chapter documents interfaces of the architecture-independent parts of the Linux kernel which are relevant to the architecture support.

4.1 Booting

```
asmlinkage void start_kernel(void);
```

The kernel's initialization/boot routine. The following steps should have been taken before calling `start_kernel()`:

- provide the necessary information for set_arch() which is called by start_kernel and initializes some global structures.
- set up `swapper_pg_dir`, establishing a virtual memory mapping for the kernel's code data and BSS
- zero-initialize the BSS

4.2 Kernel Entry/Exit

System call and fault handlers expect that `current` is a pointer to the invoker's `struct task_struct` (declared in `<linux/sched.h>`):

(Actually, `current` is `#define`d to be `current_set[smp_processor_id()]`, but exception handlers don't care about that.)

4.2.1 System Calls

System calls expect a stack layout as defined by `struct pt_regs` (see section `ptrace.h` (1.16)). This also makes sure that system calls find their parameters on the stack.

```
extern struct task_struct *current;
```

The architecture support must identify the appropriate system call handler to invoke. There is no architecture-independent system call dispatch facility.

When a system call returns, the architecture must perform housekeeping (see section Housekeeping (4.3)).

4.2.2 Page Fault Handling

The page fault handler has available the following general kernel interfaces (declared in <linux/mm.h>):

```
static inline struct vm_area_struct * find_vma (struct task_struct * task,
        unsigned long addr);
static inline int expand_stack(struct vm_area_struct * vma,
        unsigned long address);
extern void do_wp_page(struct task_struct * tsk,
        struct vm_area_struct * vma, unsigned long address,
        int write_access);
extern void do_no_page(struct task_struct * tsk,
        struct vm_area_struct * vma, unsigned long address,
        int write_access);
```

It should do approximately the following:

- Look up the corresponding vm_area_struct using find_vma(). If no corresponding region can be found, fail, sending SIGSEGV.
- Do various sanity checks on the address; the logic for this can copied from arch/i386/mm/fault.c.
- If the region is a stack area, try to expand the stack using expand_stack(vma, address). If that returns a non-zero value, fail.
- If the page fault was a write-protection fault, invoke

  ```
  do_wp_page(current, vma, address, error_code);
  ```

  If, on the other hand, the fault was a page-not-present fault, invoke

  ```
  do_no_page(current, vma, address, error_code);
  ```

  error_code should be non-zero if the fault occured in user mode, and zero, if in kernel mode.
- invoke kernel housekeeping (see section Kernel Housekeeping (4.3)).

4.2.3 Other Faults

Other faults should be handled as appropriate, often by just sending signals to the process. All exceptions should invoke kernel housekeeping after having been handled (see section Kernel Housekeeping (4.3)).

4.2.4 Interrupts

The kernel exports the following interfaces relevant to interrupt entry and exit:

```
extern unsigned long intr_count;    /* in <linux/sched.h> */
```

(In <linux/sched.h>.) The number of nested interrupts the kernel is processing: Must be incremented at every interrupt kernel entry, and decremented at exit. The scheduler (schedule()) must only be called if intr_count == 0. See section 4.3 for more information on how this flag is used.

```
extern struct kernel_stat kstat;
```

(In <linux/kernel_stat.h>.)

```
void add_interrupt_randomness(int irq);
```

(In <linux/random.h>.)

Before invoking the interrupt handlers, the architecture support should do the following:

- Acknowledge the interrupt to the hardware.
- `intr_count++;`
- For "slow" interrupts, re-enable interrupts, but leave masked the interrupt currently handled.
- `kstat.interrupts[irq]++;`
- Before invoking any "slow" handler installed with the `SA_SAMPLE_RANDOM` flag, call:

  ```
  add_interrupt_randomness(irq);
  ```

When all handlers have been executed, do the following:

- Disable interrupts.
- Unmask the handled interrupt.
- `intr_count--;`
- For "fast" interrupts, return to the interrupted context.
- For "slow" interrupts, invoke the kernel's housekeeping routine (see section Kernel Housekeeping (4.3).

For information on registering interrupts see section `sched.h` (2.4). For information on interrupt handler implementation, see the *Kernel Hacker's Guide (http://linuxwww.db.erau.edu/KHG/)*.

4.3 Kernel Housekeeping

The architecture support must do some housekeeping. This is usually implemented in a routine called `ret_from_system_call()`. This housekeeping routine should be invoked every time a system call or a slow interrupt handler (see section 4.2.4) returns.

It should do the following:

1. Check if it is being called from a nested interrupt (`intr_count != 0`). If so, return immediately to the interrupted process.
2. Check if any interrupt bottom halfs need to be executed (`(bh_mask & bh_active) != 0`). If so, handle bottom halfs (see section 4.3.1). Then restart this step.
3. Was the code segment we're returning to supervisor (i.e., system call from supervisor mode, or interrupted system call)? If so, return at this point.
4. Check if the `need_resched` flag has been set. If so, call `schedule()` (see section 4.3.2), then restart the whole housekeeping routine.
5. Check if the process we're going to return to has any signals pending. If so, handle the signals now (see section 4.3.3).

4.3.1 Bottom Half Handling

The bottom halfs are handled by this kernel routine (declared in `<linux/interrupt.h>`):

```
extern unsigned long bh_active;
extern unsigned long bh_mask;

asmlinkage void do_bottom_half(void);
```

As long as `(bh_mask & bh_active) != 0`, the housekeeping routine should execute code along the following lines:

```
intr_count++;
save_flags(flags);
sti();
do_bottom_half();
restore_flags(flags);
intr_count--;
```

4.3.2 Scheduling

The kernel exports its scheduling routine to the architecture support (declared in `<linux/sched.h>`):

```
extern /* volatile */ int need_resched;

asmlinkage void schedule(void);
```

The housekeeping routine should call the `schedule()` routine whenever the `need_resched` flag is true (this flag is set by the timer interrupt routine). `schedule()` will select some process which should be executed next, calls `switch_to()` (see section `system.h` (1.20)) to switch to it, and returns.

The `schedule()` routine can also be called from system calls in order to block the current process. It must only be called if the `intr_count` flag (see section 4.2.4) is zero, i.e., it can not be used from interrupt handlers (neither top nor bottom half).

4.3.3 Signal Handling

If `(current->signal & ~(current->blocked)) != 0`, the housekeeping routine must handle signals before returning to the task. The logic of doing this is actually quite complex and is best copied from the `do_signal()` routine in `arch/i386/kernel/signal.c`.

The architecture-dependant part is posting signals not handled kernel-internally, that is, actually invoking signal handlers in the process by pushing signal frames on the process' stack.

Chapter 5

TODO

- Kernel assumptions about (that is, kernel interfaces that want to be invoked at)
  - system calls (`sys_*()`)
  - other traps and interrupts (bottom half stuff)
  - page faults (`do_rw_page()`, `do_no_page()`)
- Device drivers