Chapter 5 Kernel Code (kernel)

5.1 Introduction

Directory linux/kernel/ contains 10 C language file and 2 assembly language file and a Makefile for manage compiling as listed in List 5-1. The files in three subdirectories will be studied in the following chapters.

This chapter mainly comments about these 13 files. Firstly, we summaris the basic functions for each source code file to gain a general understanding of them, then we study each file one by one.

List 5-1 files in linux/kernel/ directory

File name	Size	Last modified time(GMT)	Description
blk_drv/		1991-12-08 14:09:29	
chr_drv/		1991-12-08 18:36:09	
math/		1991-12-08 14:09:58	
Makefile	3309 bytes	1991-12-02 03:21:37	m
asm.s	2335 bytes	1991-11-18 00:30:28	m
exit.c	4175 bytes	1991-12-07 15:47:55	m
fork.c	3693 bytes	1991-11-25 15:11:09	m
mktime.c	1461 bytes	1991-10-02 14:16:29	m
panic.c	448 bytes	1991-10-17 14:22:02	m
printk.c	734 bytes	1991-10-02 14:16:29	m
sched.c	8242 bytes	1991-12-04 19:55:28	m
signal.c	2651 bytes	1991-12-07 15:47:55	m
sys.c	3706 bytes	1991-11-25 19:31:13	m
system_call.s	5265 bytes	1991-12-04 13:56:34	m
traps.c	4951 bytes	1991-10-30 20:20:40	m
vsprintf.c	4800 bytes	1991-10-02 14:16:29	m

5.2 Main Functions

The source code in this directory can be seperated into three classes. The first one contains files used to deal with interrupts generated by CPU execption, hardware request and software instructions. The second class is the files for system call services. The last class contains process generating, scheduling, exiting and some other assistant code files. The relationship of these files is demonstrated in Figure 5-1.

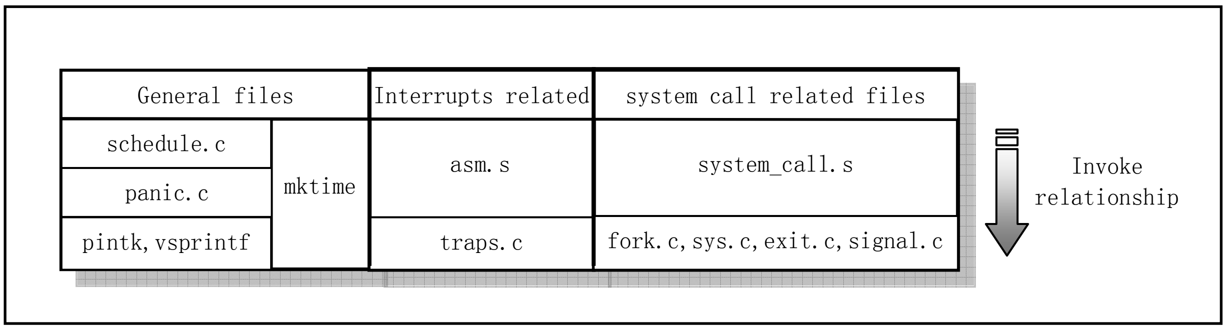

Figure 5-1 The relationship of files in kernel/ directory

5.2.1 Interrupt processing

There are two files related to interrupts handling: asm.s and traps.c. asm.s includes most part of assembly codes for hardware execption handling, while traps.c includes most C functions that called from the interrupt assembly codes. This C functions sometimes called Bottom-halfs of the interrupt handling part. In addition, several other interrupt handling functions are implemented in system_call.s and mm/page.s file.

An interrupt gate or trap gate points indirectly to a procedure which will execute in the context of the currently executing task. The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT. The offset field of the gate points to the beginning of the interrupt or exception handling procedure. Just as with a control transfer due to a CALL instruction, a control transfer to an interrupt or exception handling procedure uses the stack to store the information needed for returning to the original procedure.

Before the CPU transfer the control to the interrupt handling procedure when a interrupt occurred, it will first push at least 12 bytes onto the stack. The other similar way as a long CALL between segments is that the CPU pushes the information onto stack of target code instead of the stack of being interrupted code. Thus, when a interrupt occurred, Linux uses kernel mode stack. In addition, an interrupt always pushes the EFLAGS register onto the stack. If there is privilege transition, for example from user to kernel privilege, CPU will also push the orignal stack pointer used by the currently executing task onto the target stack as shown in Figure 5-2.

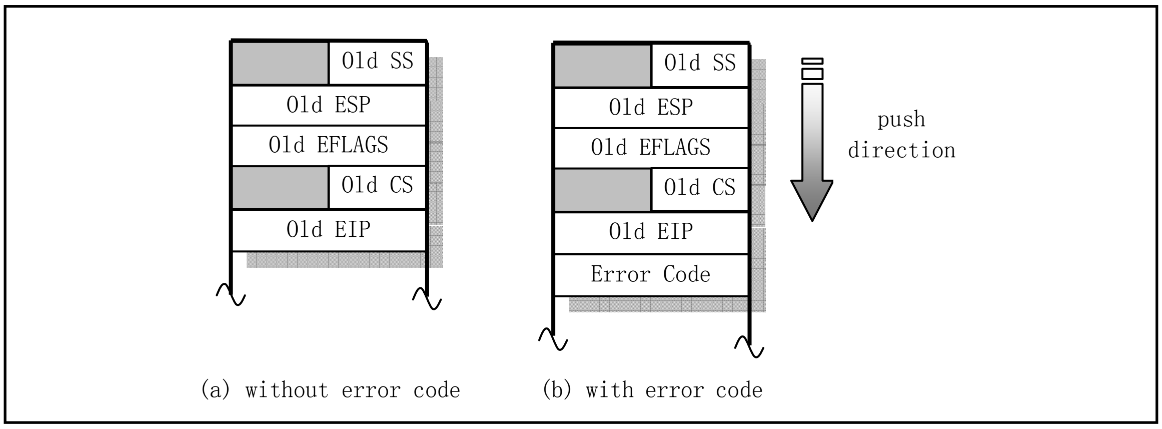

Figure 5-2 Stack layout after exception of interrupt with privilege transition

asm.s program is used to handle Intel reserved interrupts 0x00-0x1f. For Linux 0.11, the programmerable interrupt controllers (PIC) were initialized to use interrupt vectors with in range of 0x20-0x2f and the corresponding interrupt handling procedures are scattered in each initialization routines of hardware drivers. The handling of Linux system call interrupt int 0x80 is in file kernel/system_call.s. For the predetermined identifiers, see the table after the program listing.

As some software exceptions that relate to a specific segment will cause CPU to generate a error code internally, program asm.s clarify the processing of all interrupts into two classes based on whether a interrupt contains a error code. But the handling procedure of them are similar as depictured in Figure 5-3.

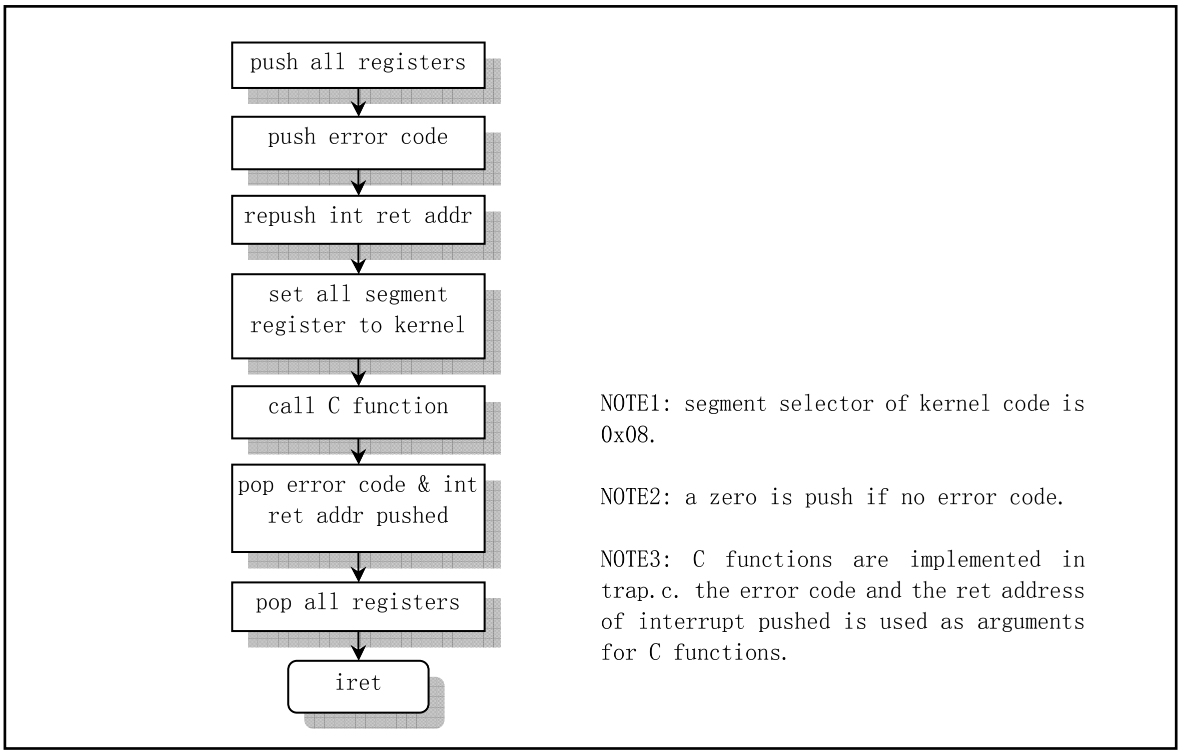
push all registers
push error code
repush int ret addr
set all segment register to kernel
call C function
pop error code & int ret addr pushed
pop all registers
iret
NOTE1: segment selector of kernel code is 0x08.
NOTE2: a zero is push if no error code.
NOTE3: C functions are implemented in trap.c. the error code and the ret address of interrupt pushed is used as arguments for C functions.

Figure 5-3 The flow chart of hardware exception (fault, traps) handling

5.2.2 System calls handling

In Linux system, application program that use the function of kernel is through the system call, that is the interrupt 0x80. The register eax contains the call function number and ebx, ecx and edx were used to store arguments if there is any. In Linux 0.11 kernel, there are 72 system calls implemented in programs system_call.s, fork.c, signal.c, sys.c and exit.c.

The way system_call.s program handling system call interrupt is similar as in asm.s program. In addition, it also contains the handling procedure of timer, hard disk and floppy drives. fork.c and signal.c are used to privide C functions for corresponding interrupt handling just like the operation of traps.c. The primary functions in fork.c is find_empty_process() and copy_process(). For the signal.c, see the discussing at the beginning of its program listing.

sys.c and exit.c program implements numerous other system calls C functions (sys_xxx()). We can properly know the using of a particular functions by its naming method. For example, a C function name begins with 'do_' is usually a general function invoked in the process of a system call; The C function begins with 'sys_' is usually only called in a particular system call interrupt. For example, do_signal() is invoked by almost all system calls in system_call.s, while sys_execve() is only invoked in a particular system call.

5.2.3 Other programs

These programs include schedule.c, mktime.c, panic.c printk.c and vsprintf.c.

schedule.c contains the most important routines: schedule(), sleep_on() and wakeup() used to switch and change the status of tasks. In addition, there are several routines related to timer handling for timer interrupt and floppy drives. mktime.c program is used to calculate the start time of kernel and only invoked in init/main.c. panic.c contains a routine panic() and used to display vital error occurred in kernel code and stop the kernel. printk.c and vsprintf.c are support programs used to display messages of the kernel.

5.3 Makefile file

5.3.1 Functions

kernel/Makefile is a configuration file used by make tools for compiling the programs in kernel/ directory, not including the programs in three subdirectories. This file is almost identical with the file listed in program 2-1 in chapter 2.

5.3.2 Comments

Program 5-1 linux/kernel/Makefile

```
#
# Makefile for the FREAX-kernel.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# (The Initial name of Linux is called FREAX an was changed to 'Linux' by the administrator
# of ftp.funet.fi)

AR      =gar    # GNU ar program used to create, modify, and extract files from archives.
AS      =gas    # GNU assembler.
LD      =gld    # GNU linker used to combine a number of object and archive files.
LDFLAGS =-s -x  # Options for gld. -s strip all symbols; -x delete all local symbols.
CC      =gcc    # GNU C language compiler.
CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
        -finline-functions -mstring-insns -nostdinc -I../include
# Options for gcc.
# -Wall show all warning messages;
# -O optimze code length and executing time;
# -fstrength-reduce optimzes codes in loop, eliminates iteration variables;
# -fomit-frame-pointer Don't keep frame pointer in a register for functions that don't need;
# -fcombine-regs Combine registers, minimum the using of register classes;
```

```
   # -mstring-insns This option is added by Linus to gcc 1.40 used to adopt string instructions
   # in i386.
   # -nostdinc Don't use header files in default directory, e.g. /usr/include/
   # -I../include Indicates the directory of header files used (../include).
16 CPP     =gcc -E -nostdinc -I../include
   # Options for C preprocessor.
   # -E only runs the C preprocessing, The output is in the form of preprocessed source code, which
   # is sent to the standard output;
17
   # The following role (L18) is used to indicate make program to use the command (L19) below
   # to compile all '.c' files into '.s' assembly files.
   # The command (L19) indicates gcc to compile C code by using the options provided by 'CFLAGS',
   # but don't run linking process (-S). Thus producing assembler files coresponding each '.c'
   # files with suffix of '.s'.
   # -o indicats the following is name of output; '$*.s' (or '$@') is automatic variable.
   # '$<' represents the first precondition, e.g. the '*.c' files here.
18 .c.s:
19         $(CC) $(CFLAGS) \
20         -S -o $*.s $<
   # The following role will compile all '.s' assembler file into '.o'object files using gas.
21 .s.o:
22         $(AS) -c -o $*.o $<
23 .c.o:                      # '*.c' files to '*.o' object files, don't do linking.
24         $(CC) $(CFLAGS) \
25         -c -o $*.o $<
26
27 OBJS  = sched.o system_call.o traps.o asm.o fork.o \     # Define Object variables OBJS.
28         panic.o printk.o vsprintf.o sys.o exit.o \
29         signal.o mktime.o
30
   # Linking to produce target object 'kernel.o' after precondition 'OBJS' is satisfied.
31 kernel.o: $(OBJS)
32         $(LD) -r -o kernel.o $(OBJS)
33         sync
34
   # The following role is used to do cleaning after once compile. When we execute 'make clean'
   # at shell's command prompt, the 'make' will run the commands below (L36-40), erasing all
   # files generated by compiler.
   # 'rm' is file deletion command. Option '-f' neglecting non-exist files and errors.
35 clean:
36         rm -f core *.o *.a tmp_make keyboard.s
37         for i in *.c;do rm -f `basename $$i .c`.s;done
38         (cd chr_drv; make clean)
39         (cd blk_drv; make clean)
40         (cd math; make clean)
41
   # The following role is used to check the dependence between each source files:
   # Use stream editor 'sed' to deal with 'Makefile' (this file), the output is a temporary
   # file 'tmp_make', in which, all lines after '### Dependencies' (begins from L51) were
   # deleted. Then, for all '.c' files in the directory, output a file name, which suffix is
   # converted to '.s' and followed by a white space character, and, afterwards, executing
   # preprocess programm (CPP) to produce a dependent line for the corresponding '.c' file as
   # illustrated in line L51-55 for exit.c file. Finally, copy the temporary file to 'Makefile'
   # to renew it.
   # Option '-M' tells the CPP to output the roles for each relating object file in make syntax.
   dep:
           sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
           (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
                   $(CPP) -M $$i;done) >> tmp_make
           cp tmp_make Makefile
           (cd chr_drv; make dep)
           (cd blk_drv; make dep)

   ### Dependencies:
   exit.s exit.o : exit.c ../include/errno.h ../include/signal.h \
     ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
     ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
     ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
     ../include/asm/segment.h
   fork.s fork.o : fork.c ../include/errno.h ../include/linux/sched.h \
     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
     ../include/asm/segment.h ../include/asm/system.h
   mktime.s mktime.o : mktime.c ../include/time.h
   panic.s panic.o : panic.c ../include/linux/kernel.h ../include/linux/sched.h \
     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
     ../include/linux/mm.h ../include/signal.h
   printk.s printk.o : printk.c ../include/stdarg.h ../include/stddef.h \
     ../include/linux/kernel.h
   sched.s sched.o : sched.c ../include/linux/sched.h ../include/linux/head.h \
     ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
     ../include/signal.h ../include/linux/kernel.h ../include/linux/sys.h \
     ../include/linux/fdreg.h ../include/asm/system.h ../include/asm/io.h \
     ../include/asm/segment.h
   signal.s signal.o : signal.c ../include/linux/sched.h ../include/linux/head.h \
     ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
     ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h
   sys.s sys.o : sys.c ../include/errno.h ../include/linux/sched.h \
     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
     ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
     ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h \
     ../include/sys/times.h ../include/sys/utsname.h
   traps.s traps.o : traps.c ../include/string.h ../include/linux/head.h \
     ../include/linux/sched.h ../include/linux/fs.h ../include/sys/types.h \
     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
     ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
   vsprintf.s vsprintf.o : vsprintf.c ../include/stdarg.h ../include/string.h
```

5.4 asm.s

5.4.1 Functions

asm.s program involves the low level codes for processing exceptions detected by CPU, includes the handling of FPU exceptions. This program is intimately related to kernel/traps.c program. Its primary handling method is to invoke the corresponding C functions to do the detail processing, display the error location and error code, and finally exit.

When start reading this piece of code, it is helpful to refer to the changes in stack as illustrated in Figure 5-4. In this figure, each line of frames represents 4 bytes. Before executes the code, the stack pointer esp points to the interrupt return address (esp0 in figure). After it pushes the address of C function do_divide_error() or other related C funtions into the stack, stack pointer moves to the location esp1 in figure. At this moment, the function's address is swapped into register eax while the value of eax is swapped into stack. After several other registers pushed into stack, the location of stack pointer is at esp2. Just before invoking the do_divide_error(), it pushes the esp0 into stack as a argument for C function, that is at the location of esp3 in figure. When returns from the C function, it adds value 8 to the current stack pointer to back to the location of esp2.

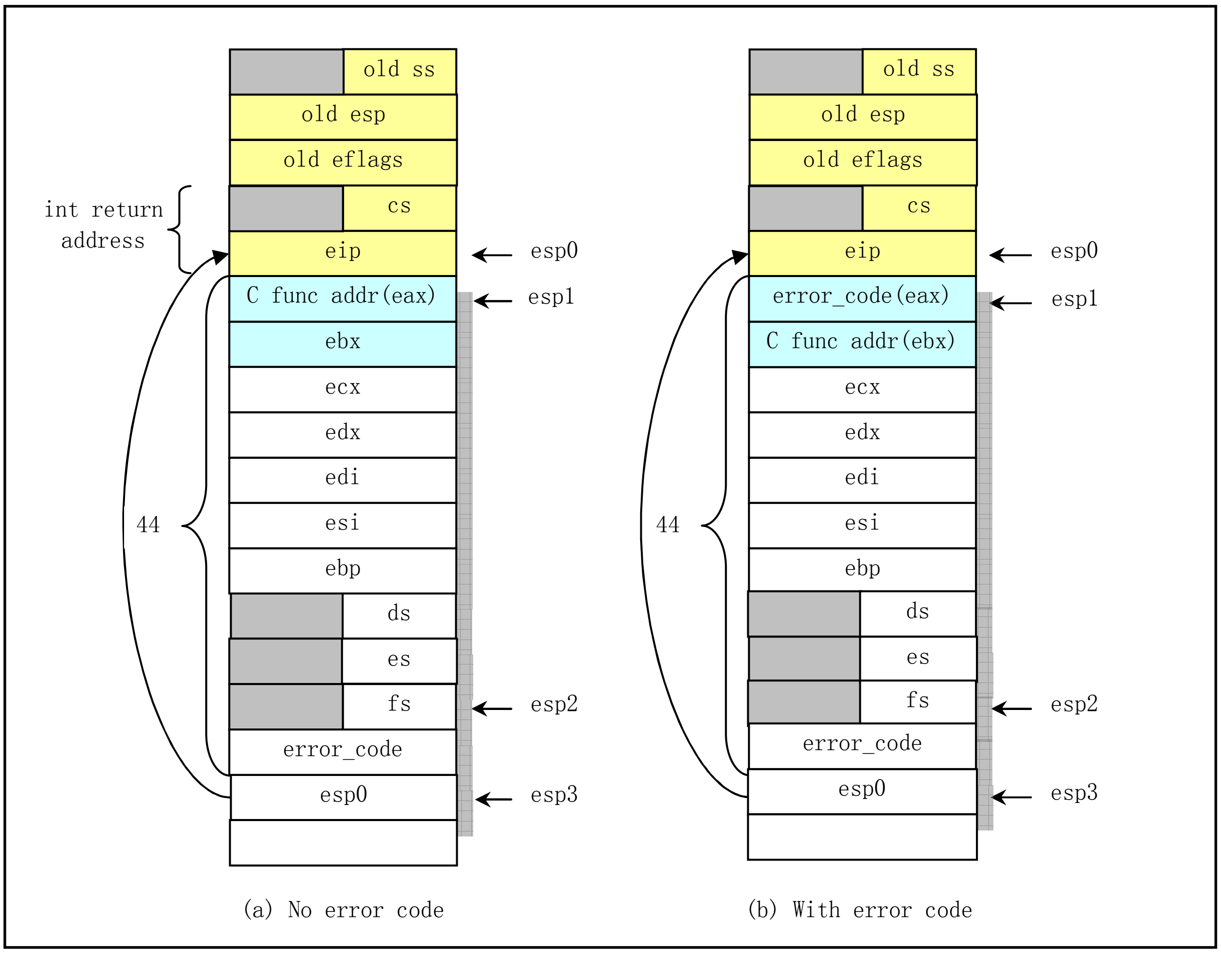

Figure 5-4 Stack map of error handling for exceptions

As stated above, the error code and the value of stack pointer when just enter the program (esp0) are used as arguments for the C function (e.g. do_divide_error()). In the program traps.c, the prototype of do_divide_error() is:

```
void do_divide_error(long esp, long error_code)
```

Therefore, this function can print out the error location and error code of the program causing exception.

5.4.2 Comments

Program 5-2 linux/kernel/asm.s

```
 1 /*
 2  *  linux/kernel/asm.s
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * asm.s contains the low-level code for most hardware faults.
 9  * page_exception is handled by the mm, so that isn't here. This
10  * file also handles (hopefully) fpu-exceptions due to TS-bit, as
11  * the fpu must be properly saved/resored. This hasn't been tested.
12  */
13
   # This program is used to handling the Intel reserved interrupts 0x00-0x16.
   # The following lines are declarations of global functions, implemented in traps.c.
14 .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
15 .globl _double_fault,_coprocessor_segment_overrun
16 .globl _invalid_TSS,_segment_not_present,_stack_segment
17 .globl _general_protection,_coprocessor_error,_irq13,_reserved
18
   # The following codes were used to handle exceptions with no error code. Refers to figure 5-4(a).
   # The label '_do_divide_error' is actually a internal name representing the coressponding C
   # function do_divide_error().
   #
   # int 0 - Procedure for handling devide by zero error. (Fault)
19 _divide_error:
20         pushl $_do_divide_error    # Push the addr of C function onto stack first.
                                      # AT&T immediate operands are preceded by '$'.
21 no_error_code:                    # Entry point for interrupts w/o error code, Refer to L55.
22         xchgl %eax,(%esp)          # Exchange the contents of stack pointed by esp and eax.
23         pushl %ebx                 # That is: addr of do_divide_error() ->eax, eax ->(esp)
24         pushl %ecx
25         pushl %edx
26         pushl %edi
27         pushl %esi
28         pushl %ebp
29         push %ds                   # 16-bit segment register will occupy 4 bytes on stack.
30         push %es
31         push %fs
   # Pushes error code and the esp of interrupt return addr as arguments for C function at L39.
32         pushl $0                   # "error code". push 0 to resemble a error code.
33         lea 44(%esp),%edx          # Push the stack pointer of interrupt return address.
34         pushl %edx
35         movl $0x10,%edx            # 0x10 is the value of kernel data segment selector.
36         mov %dx,%ds                # Prepare segment register to point to kernel
37         mov %dx,%es
38         mov %dx,%fs
```

```
        call *%eax                  # Invokes the C function (e.g. do_divide_error()).
                                    # Note that AT&T absolute (as opposed to PC relative)
                                    # jump/call operands are prefixed by '*'
        addl $8,%esp                # Get rid of two values on stack.
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax                   # Pop the original eax.
        iret

# int1 -- Entry point for debug interrupt. (Fault/Trap)
# Raised when the T flag of eflags is set or when the address of an instruction or operand
# falls within the range of an active debug register
_debug:
        pushl $_do_int3         # Push the address of do_debug().
        jmp no_error_code

# int2 -- Entry point for nonmaskable interrupts (NMI). (Trap)
_nmi:
        pushl $_do_nmi
        jmp no_error_code

# int3 -- Entry point for breakpoint interrupt. (Trap)
# Caused by an int3 instruction.Usually inserted by a debugger.
_int3:
        pushl $_do_int3
        jmp no_error_code

# int4 -- Entry point for overflow issued interrupt. (Trap)
# Raised when the OF (overflow) flag of eflags is set.
_overflow:
        pushl $_do_overflow
        jmp no_error_code

# int5 -- Entry point for bounds check interrupt. (Fault)
# A bound instruction is executed with the operand outside of the valid address bounds.
_bounds:
        pushl $_do_bounds
        jmp no_error_code

# int6 -- Entry point for invalid opcode interrupt. (Fault)
# The CPU execution unit has detected an invalid opcode.
_invalid_op:
        pushl $_do_invalid_op
        jmp no_error_code

# int9 -- Entry point for coprocessor error. (Fault)
# Problems with the external mathematical coprocessor.
_coprocessor_segment_overrun:
        pushl $_do_coprocessor_segment_overrun
        jmp no_error_code

# int15 - Entry point for interrupt reserved by Intel.
_reserved:
        pushl $_do_reserved
        jmp no_error_code

# int45 -- (= 0x20 + 13) Coprocessor generated interrupt, configured by Linux in PIC.
# When copprocessor finishes an operation, it will issue IRQ 13 to tell CPU the state.
# During the operation of 80387, the CPU will wait for its finish.
_irq13:
        pushl %eax
        xorb %al,%al
# By write to I/O port 0xf0, this interrupt handler will remove the CPU's extension of BUSY
# signal and reactivate the 80387's pin of coprocessor extended request (PEREQ).
# The use of an interrupt for a copprocessor is to guarantee that an exception from a
# coprocessor instruction will be detected and INTR 13 will be serviced prior to the
# execution of any further 80387 instructions.
        outb %al,$0xF0
        movb $0x20,%al
        outb %al,$0x20          # Send End of Interrupt (EOI) signal to 8259A master chip.
        jmp 1f                  # Delay a while.
1:      jmp 1f
1:      outb %al,$0xA0          # Send EOI signal to 8259A slave chip.
        popl %eax
        jmp _coprocessor_error  # _coprocessor_error is at L131 in kernel/system_call.s.

# When processing the following interrupts, CPU will push the error code onto stack after the
# the interrupt return address. Therefore, we should pop up the error code when return.
# Refers to figure 5-4(b).
# int8 -- Entry point for double fault interrupt. (Abort)
# Normally, when the CPU detects an exception while trying to call the handler for a prior
# exception, the two exceptions can be handled serially. In a few cases, however, the processor
# cannot handle them serially, so it raises this exception.
_double_fault:
        pushl $_do_double_fault         # Push the pointor of C function onto stack.
error_code:
        xchgl %eax,4(%esp)              # error code <-> %eax, old eax is saved onto stack.
        xchgl %ebx,(%esp)               # &function <-> %ebx, old ebx is saved onto stack.
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl %eax                      # error code
        lea 44(%esp),%eax               # offset
        pushl %eax
        movl $0x10,%eax                 # Setup kernel data segment register.
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        call *%ebx                      # Invoke the C function. Arguments are on the stack.
        addl $8,%esp                    # Get gid of arguments on the stack.
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        iret

# int10 -- Entry point for invlid task status segment (TSS). (Fault)
# The CPU has attempted a context switch to a process having an invalid Task State Segment.
_invalid_TSS:
        pushl $_do_invalid_TSS
        jmp error_code

# int11 -- Entry point for segment not present interrupt. (Fault)
# A reference was made to a segment not present in memory.
_segment_not_present:
        pushl $_do_segment_not_present
        jmp error_code

# int12 -- Entry point for stack segment error interrupt. (Fault)
# The instruction attempted to exceed the stack segment limit, or the segment identified by
# ss is not present in memory.
_stack_segment:
        pushl $_do_stack_segment
        jmp error_code

# int13 -- Entry point for general protection interrupt. (Fault).
# One of the protection rules in the protected mode of the 80386 has been violated.
_general_protection:
        pushl $_do_general_protection
        jmp error_code

# int7 --  _device_not_available is at L148 in kernel/system_call.s file.
# int14 -- _page_fault) is at L14 in mm/page.s file.
# int16 -- _coprocessor_error is at L131 in kernel/system_call.s file.
# int 0x20 -- _timer_interrupt is at L176 in kernel/system_call.s file.
# int 0x80 -- _system_call is at L80 in kernel/system_call.s file.
```

5.4.3 Information

5.4.3.1 Intel reserved interrupts

Table 5-1 presents the difinition of interrupt vectors reserved by Intel.

Table 5-1 Interrupt vectors reserved by Intel

Int No.	Name	Type	Signal	Description
0	Devide error	Fault	SIGFPE	Occurred during a divide instruction when the divisor is zero.
1	Debug	Fault/ Trap	SIGTRAP	Raised when the T flag of eflags is set or when the address of an instruction or operand falls within the range of an active debug register.
2	nmi	Hardware		Used for nonmaskable interrupts (NMI)
3	Breakpoint	Trap	SIGTRAP	The int 3 instruction causes this trap.
4	Overflow	Trap	SIGSEGV	This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set.
5	Bounds check	Fault	SIGSEGV	This fault occurs when the processor finds that the operand exceeds the specified limits while executing a bounds instruction.
6	Invalid Opcode	Fault	SIGILL	This fault occurs when an invalid opcode is detected by the CPU execution unit.
7	Device not available	Fault	SIGSEGV	This exception occurs in either of two conditions: (a) The CPU encounters an ESC instruction, and the EM (emulate) bit of CR0 is set; (b) The CPU encounters either the WAIT instruction or an ESC instruction, and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.
8	Double fault	Abort	SIGSEGV	When the CPU detects an exception while trying to invoke the handler for a prior exception, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals this exception instead.
9	Coprocessor segment overrun	Abort	SIGFPE	Raised in protected mode if the 80386 detects a page or segment violation while transferring the middle portion of a coprocessor operand to the NPX.
10	Invalid TSS	Fault	SIGSEGV	Occurs if during a task switch the new TSS is invalid.
11	Segment not present	Fault	SIGBUS	A reference was made to a segment not present in memory.
12	Stack segment	Fault	SIGBUS	The instruction attempted to exceed the stack segment limit, or the segment identified by ss is not present in mem.

13	General protection	Fault	SIGSEGV	All protection violations that do not cause anohter exception cause this exception.
14	Page fault	Fault	SIGSEGV	This exception occurs when CPU finds the page is not present in memory.
15	Reserved			
16	Coprocessor error	Fault	SIGFPE	The CPU reports this exception when it detects a signal from the 80387 on the 80386's ERROR# input pin.

5.5 traps.c

5.5.1 Functions

traps.c program mainly includes the C functions invoked by the low level code in asm.s file. These C functions is usually used to show the error location and error code of the exception. In the current Linux system, these C functions is usually called the bottom halfs, because the quantity of their code or processing speed is relately slow, thus, they may be interrupted by other code in the kernel.

The die() function is used to display detailed error information about the exception on the console. The trap_init() function at the end of traps.c program, used to initialize interrupt vectors for the CPU or hardware exceptionsis and enable the interrupt requests, is invoked in the init/main.c program. When read this program, it's better to refer to asm.s program at the same time.

5.5.2 Comments

Program 5-3 linux/kernel/traps.c

```
/*
 *  linux/kernel/traps.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */
#include <string.h>       // Declares one type and several functions for manipulating strings.

#include <linux/head.h>   // define a simple structure of segment descript and a few constans.
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/segment.h>
```

```
#include <asm/io.h>

// The following lines define three inline assembly macros. A compound statement in parentheses
// may appear inside an expression in GNU C. This allows you to declare variables within an
// expression. The last variable (__res) is the output value of the macro. Refers to the
// descriptions for inline assembly instructions.
//
// Get one byte from address 'addr' in segment 'seg'.
// Arguments: seg - segment selector; addr - offset in the segment.
// %0 - eax (__res); %1 - eax (seg); %2 - offset (*(addr)).
#define get_seg_byte(seg,addr) ({ \
register char __res; \
__asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
        :"=a" (__res):"" (seg),"m" (*(addr))); \
__res;})

// Get a long word (4-byte) from address 'addr' in segment 'seg'.
// Arguments: seg - segment selector; addr - offset in the segment.
// %0 - eax (__res); %1 - eax (seg); %2 - offset (*(addr)).
#define get_seg_long(seg,addr) ({ \
register unsigned long __res; \
__asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
        :"=a" (__res):"" (seg),"m" (*(addr))); \
__res;})

// Get the value of fs register (selector).
// %0 - eax (__res).
#define _fs() ({ \
register unsigned short __res; \
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

int do_exit(long code);                      // At L102 in kernel/exit.c

void page_exception(void);                   // page_fault. At L14 in mm/page.s

// The following lines define the prototypes of interrupt handling procedure. They were
// used to setup the corresponding interrupt gate descriptors in function trap_init().
void divide_error(void);                     // int0 (kernel/asm.s,L19)。
void debug(void);                            // int1 (kernel/asm.s,L53)。
void nmi(void);                              // int2 (kernel/asm.s,L57)。
void int3(void);                             // int3 (kernel/asm.s,L61)。
void overflow(void);                         // int4 (kernel/asm.s,L65)。
void bounds(void);                           // int5 (kernel/asm.s,L69)。
void invalid_op(void);                       // int6 (kernel/asm.s,L73)。
void device_not_available(void);             // int7 (kernel/system_call.s,L148)。
void double_fault(void);                     // int8 (kernel/asm.s,L97)。
void coprocessor_segment_overrun(void);      // int9 (kernel/asm.s,L77)。
void invalid_TSS(void);                      // int10 (kernel/asm.s,L131)。
void segment_not_present(void);              // int11 (kernel/asm.s,L135)。
void stack_segment(void);                    // int12 (kernel/asm.s,L139)。
void general_protection(void);               // int13 (kernel/asm.s,L143)。
void page_fault(void);                       // int14 (mm/page.s,L14)。
void coprocessor_error(void);                // int16 (kernel/system_call.s,L131)。
void reserved(void);                         // int15 (kernel/asm.s,L81)。
void parallel_interrupt(void);               // int39 (kernel/system_call.s,L280)。
void irq13(void);                            // int45 (kernel/asm.s,L85)。

// This function is used to print the interrupt name, error code, and the messages related to
// the calling process including eip, eflags, esp, base address and limits of segments,
// process pid, task number and 10 bytes opcodes. If the stack is in user data segment, then
// the 16 bytes of stack is printed too.
static void die(char * str,long esp_ptr,long nr)
{
        long * esp = (long *) esp_ptr;
        int i;

        printk("%s: %04x\n\r",str,nr&0xffff);
        printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
                esp[1],esp[0],esp[2],esp[4],esp[3]);
        printk("fs: %04x\n",_fs());
        printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
        if (esp[4] == 0x17) {
                printk("Stack: ");
                for (i=0;i<4;i++)
                        printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
                printk("\n");
        }
        str(i);                 // Get task number of the current process.(L159, sched.h)
        printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
        for(i=0;i<10;i++)
                printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
        printk("\n\r");
        do_exit(11);            /* play segment exception */
}

// The following functions are invoked by relating interrupt handlers.
void do_double_fault(long esp, long error_code)
{
        die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
        die("general protection",esp,error_code);
}

void do_divide_error(long esp, long error_code)
{
        die("divide error",esp,error_code);
}

void do_int3(long * esp, long error_code,
                long fs,long es,long ds,
                long ebp,long esi,long edi,
                long edx,long ecx,long ebx,long eax)
```

```
{
        int tr;

        __asm__("str %%ax":"=a" (tr):"" (0)); // Get task register's value into variable tr.
        printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
                eax,ebx,ecx,edx);
        printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
                esi,edi,ebp,(long) esp);
        printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
                ds,es,fs,tr);
        printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

void do_nmi(long esp, long error_code)
{
        die("nmi",esp,error_code);
}

void do_debug(long esp, long error_code)
{
        die("debug",esp,error_code);
}

void do_overflow(long esp, long error_code)
{
        die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
        die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
        die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
        die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
        die("coprocessor segment overrun",esp,error_code);
}

void do_invalid_TSS(long esp,long error_code)
{
        die("invalid TSS",esp,error_code);
}

```

```c
void do_segment_not_present(long esp,long error_code)
{
        die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
        die("stack segment",esp,error_code);
}

void do_coprocessor_error(long esp, long error_code)
{
        if (last_task_used_math != current)
                return;
        die("coprocessor error",esp,error_code);
}

void do_reserved(long esp, long error_code)
{
        die("reserved (15,17-47) error",esp,error_code);
}

// This is the initialization function of interrupt exception used to setup various interrupt
// gates.
// The main difference between set_trap_gate() and set_system_gate() is that the former's
// privilege level is set to 0, and the later's is set to 3. Thus the handler of breakpoint
// int3, overflow and bounds interrupt can be invoked by any programs. This two macros were
// implemented in include/asm/system.h file.
void trap_init(void)
{
        int i;

        set_trap_gate(0,&divide_error);
        set_trap_gate(1,&debug);
        set_trap_gate(2,&nmi);
        set_system_gate(3,&int3);       /* int3-5 can be called from all */
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_trap_gate(8,&double_fault);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_trap_gate(14,&page_fault);
        set_trap_gate(15,&reserved);
        set_trap_gate(16,&coprocessor_error);
// The following loop first batch installing the vectors of int 17-48 to be the handler of
// 'reserved'. Each hardware initial routine will reinstall its handler descriptor later.
        for (i=17;i<48;i++)
                set_trap_gate(i,&reserved);
```

```
   // Installing the trap gates of copprocessor and parallel device. Enabling the interrupt
   // request of PIC chips.
        set_trap_gate(45,&irq13);           // Trap gate of coprocessor.
        outb_p(inb_p(0x21)&0xfb,0x21);      // Enable IRQ2 of 8259A master chip.
        outb(inb_p(0xA1)&0xdf,0xA1);        // Enable IRQ13 of slave chip.
        set_trap_gate(39,&parallel_interrupt);  // Trap gate of parallel device.
   }

```

5.5.3 Information

5.5.3.1 GNU gcc Inline assembly

This is the first time we encounter the inline assembly instructions in C language. Because we have very few chance to use it in normal application programming, thus, here, we present a simple description about its syntax, enough to get the ability to understand the kernel source code. Refers to GNU gcc manual or an article "Using Inline Assembly with gcc" for a detailed description of it.

The following is the basic format of the inline assembly with input and output arguments:

```
asm("statements"
    : output_registers
    : input_registers
    : clobbered_registers);
```

"statements" is the location you write assembly instructions, 'output_registers' (or called output operands) indicates where the output values were in the registers after finished executing the assembly instructions. Here, these registers are associated with C variables or an address in memory seperately. 'input_registers' (or called input operands) represents the input values each indicated register has before start running assembly instructions. They may associated with C expressions or constant values. 'clobbered_registers' is a set of registers used to tell gcc can no longer count on the values it loaded into these registers. When we use inline assembly, the last three lines of the format can be omited if not used.

Let's explain the using method with an example. The following lines are a piece of code selected begin from line 22 from previous program (Program 5-3). We have rearranged it to have a better layout.

```
01  #define get_seg_byte(seg,addr) \
02  ({ \
03  register char __res; \
04  __asm__("push %%fs;  \
05           mov %%ax,%%fs;  \
06           movb %%fs:%2,%%al;  \
07           pop %%fs"  \
08           :"=a" (__res)  \
09           :"" (seg),"m" (*(addr)); \
10  __res;})
```

This piece of code defines a macro function using the inline assembly. Usually the most convenient way to use inline assembly is to encapsulate them in macros. A compound statement in parentheses (sentences in bracket parentheses) may appear inside an expression in GNU C. This allows us to declare variables within an expression. The value of the variable __res at the last line is the output value of the expression.

The first line (L01) defines macro’s identifier with parameters givien in the parentheses. L02 is the beginning of a compound statement. L03 declares a register variable ’__res’ in the compound statement. The ’__asm__’ at L04 starts the inline assembly statements. Instructions within L04-07 are AT&T format assembly sentences.

L08 is the output operand section, contains one operand here. The ’__res’ is the C expression of the output operand. Output operand expressions must be lvalues and write-only. ’″a″’ is the operand constraint, saying that a eax register is required. The ’=’ in ’=a’ indicates that the operand is an output. All output operands’ constraints must use ’=’. Each operand is described by an operand-constraint string followed by the C expression in paretheses. A colon separates the assembler statements from the first output operand, and another separates the last output operand from the first input operand, if any. Commas are used to separate output operands and separate input operands.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

L09 is the input operand section, contains two operands here. The ’seg’ is the C expression for the first input operand while the ’*(addr)’ is that of the second. The ’″″’ is an ellipsis of ’″0″’ for input operand 1, means that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand. In order to refer to the operands in the inline assembler statements, all operands are ordered begin from zero (%0) with the first operand in output operands or input operands if output operands are not existed. For example the input operand ’*(addr)’ is refered to with ’%2’ in the assembler statements. The ’″m″’ is the constraint for the second input operand, means that the second operand is a memory effective address.

The input operands must be read-only. When the assembler instruction has a read-write operand, you must logcally split its function into two separate operands, one input operand and one write-only output operand.

Now, let’s take a look for the meaning of assembler statements at L04-07. The first instruction push the segment register fs onto stack; The second instruction, then, store the eax into fs, that is store the value of ’seg’ into fs; The third instruction save the value at ’fs:(*(addr))’ into register al. When finished executing the assembler instructions, this value is stored into variable ’__res ’ in according to the output operand constraint. So the whole macro means: ’seg’ indicates a segment in memory, ’addr’ is an offset address in the segment. The macro returns one byte value pointed by ’seg’ and ’addr’ in memory.

Notice that in the inline assembly, you must prefix registers with ’%%’ instead of just ’%’. Because as GCC parses along for ’%0’ and ’%1’ and so on, it would interpret ’%edx’ as a ’%e’ parameter, and that is non-existent, and cause an error.

Let’s see another example below.

```
01  asm("cld\n\t"
02      "rep\n\t"
03      "stol"
04      : /* no outputs */
05      : "c"(count-1), "a"(fill_value), "D"(dest)
06      : "%ecx", "%edi");
```

The '\ns' and '\t's are there so the '.s' file that gcc generates and hands to gas comes out right when you have got multiple statements per asm. It's really meant for issuing instructions for which there is no equivalent in C and don't touch the registers.

L01-03 are the normal assembler instructions used to clear the direction of string operating and repeat saving value. L04 indicates that is inline assembly has no output operand. L05 is the input operands: the value of 'count-1' will be stored in register ecx (operand constraint is "c"), eax with 'fill_value' and edi with 'dest'. Why make GCC do it instead of doing it ourselves? Because GCC, in its register allocating, might be able to arrange for, say, 'fill_value' to already be in eax. If this is in a loop, it might be able to preserve eax through the loop, and save a movl once per loop. L06 specifies GCC that it can no longer count on the values it loaded into ecx or edi to be valid. This doesn't mean they will be reloaded for certain. This is the clobberlist.

Seem strange? Well, it really helps when GCC do optimizing, when GCC can know exactly what you are doing with the registers before and after. It folds your assembly code into the code it generates and then optimizes. Table 5-2 is a list of operand constraints that we often use.

Table 5-2 Operand constraints for inline assembly

Code	Description	Code	Description
a	eax	m	memory effective address
b	ebx	o	memory address with offset
c	ecx	I	constant value (0 to 31).
d	edx	J	constant value (0 to 63).
S	esi	K	constant value (0 to 255).
D	edi	L	constant value (0 to 65535).
q	Dynamicaly addressed register. (eax, ebx, ecx or edx)	M	constant value (0 to 3).
r	Dynamicaly addressed register. (eax, ebx, ecx, edx, esi, edi)	N	one byte constant value (0 to 255).
g	general register (eax, ebx, ecx, edx or variable in memory).	O	constant value (0 to 31).
A	eax and edx combined into a 64-bit integer (use long longs)	n	immediate value known at compile time.

Its possible to let GCC choose the registers for you as in the following example:

```
01  asm("leal (%1, %1, 4), %0"
02        : "=r"(y)
03        : "0"(x));
```

The first assembly statement (L01) "leal (r1, r2, 4), r3" means r1 + r2*4 --> r3. This example multiplies 'x' by 5 really quickly. Now, we could have specified, say eax. But unless we really need a specific register (like when using rep movsl or rep stosl, which are hardcoded to use ecx, edi, and esi), why not let GCC pick an available one? So when GCC generates the output code for GAS, %0 will be replaced by the register it picked.

Notice that if you want to prevent an asm instruction from beging deleted, moved or combined by GCC, you can write keyword 'volatile' after the 'asm'. For example:

```
asm volatile (...);
```

The following example is a little longer one selected from file include/string.h.

```
//// Compare first count bytes between string str1 and str2.
// Arguments: cs - Str1, ct - Str2, count - Compared byte count.
// %0 - eax(__res) ret value; %1 - edi(cs) point to str1; %2 - esi(ct) point to str2; %3 - ecx(count)。
// Return 1 if str1 > str2; 0 if str1=str2; -1 if str1 < str2.
extern inline int strncmp(const char * cs,const char * ct,int count)
{
register int __res ;                  // register variable.
__asm__("cld\n"                       // Clear direction bit in eflags.
         "1:\tdecl %3\n\t"            // count--。
         "js 2f\n\t"                  // if count < 0 then jump forward to label 2.
         "lodsb\n\t"                  // Get str2's char ds:[esi] into al and esi++
         "scasb\n\t"                  // Compare al and str1's char es:[edi] and edi++
         "jne 3f\n\t"                 // Forward jump to label 3 if not equal.
         "testb %%al,%%al\n\t"        // a NULL char ?
         "jne 1b\n"                   // Backward jump to label 1 if not a NULL char.
         "2:\txorl %%eax,%%eax\n\t"   // Clear eax and return if tt's a NULL char.
         "jmp 4f\n"                   // Forward jump to label 4 and return.
         "3:\tmovl $1,%%eax\n\t"      // eax = 1
         "jl 4f\n\t"                  // Return 1 if str2 < str1,
         "negl %%eax\n"               // else return -1.
         "4:"
         :"=a" (__res):"D" (cs),"S" (ct),"c" (count)
         :"si","di","cx");
return __res;                        // return result.
}
```

5.6 system_call.s

5.6.1 Function

In the Linux system, user uses interrupt int 0x80 and the function number in register eax to invoke the function provided by kernel. These operating system internal functions are called the system call functions. Usually, the application program does not invoke the system call interrupt directly. It uses the relating interfacing function provided by general library such as libc.a. For example, when creating a new process, it usually uses function fork() in the library. The function fork() in library will implement the procedure of doing int 0x80 and return the result to user.

For all system calls, the kernel lists the function pointers in the table sys_call_table[] (in include/linux/sys.h) ordered by the function number. In the procedure of int 0x80, it invokes the system function based on the function number provied by the user in eax.

The system_call.s program is mainly used to implement the interface of system calls with user application program and detect and process the signals of the calling process. In addition, two system functions (sys_execve for executing a program and sys_fork for forking a new process) and several interrupt handling procedures are also implemented in this program. The interrupt handling procedures includes coprocessor error (int 16), device not found (int 7), timer interrupt (int 32), hard disk interrupt (int 46) and floppy device interrupt (int 38).

For the software interrupts (system calls, coprocessor_error, device_not_available), their handling procedures are basically identical: push some arguments for the C functions to call, invoke the C function, when return, check the signal map of the current process and process the siganl with the lowest value and reset the siganl processed. The C functions for all system calls are scattered in the kernel source tree. They are assembled to be a function pointer table in header file include/linux/sys.h.

For the hardware interrupt requests, the handling procedure first sends 'end of interrupt' (EOI) signal to the 8259A chip and then invokes the corresponding C functions. The handling of timer interrupt also needs to process the signal map of current process.

The handling procedure of system call (int 0x80) in this program can be considered as a interface to the real processing procedures in each corresponding C function (the bottom halfs). At the beginning of this program, it first checks the validity of the system call number in register eax and then saves all the registers used. The kernel uses the segment register ds and es for kernel data segment, fs for user data segment as default. Afterwards, it invokes the relating C functions according to the address table of C functions and saves the return value onto the stack when returns from the C funtion.

After that, it checks the state of the calling process. If the operation of the relating C function has changed the state of the calling process or the time for the calling process was used up (counter ==0), then it invokes the schedule function to switch to the other process. Although it uses 'jmp _schedule' instruction to execute the schedule function, the CPU control will be passed to here finally after the kernel switches back to the current process. Because the program pushed the return address (ret_from_sys_call) onto the stack before jump to the schedule function.

The code begin from label 'ret_from_sys_call' is used to do some ending operation for this interrupt execution. It checks the current process's type to decide if to end the interrupt handling immediately. If the process is a kernel process (e.g. process 0 or 1), it ends the handling right away. Otherwise it will handle the sigals of the process further more. If the process does have siganls based on its signal map, it will handling the signal by calling function do_signal().

At the end, the system_call.s program restores the registers saved and exits from this interrupt handling procedure. If the process has signal, then the control will pass to the signal handling function first, and after that, back to the calling program.

The flow chart of the system call interrupt processing is illustrated in Figure 5-5.

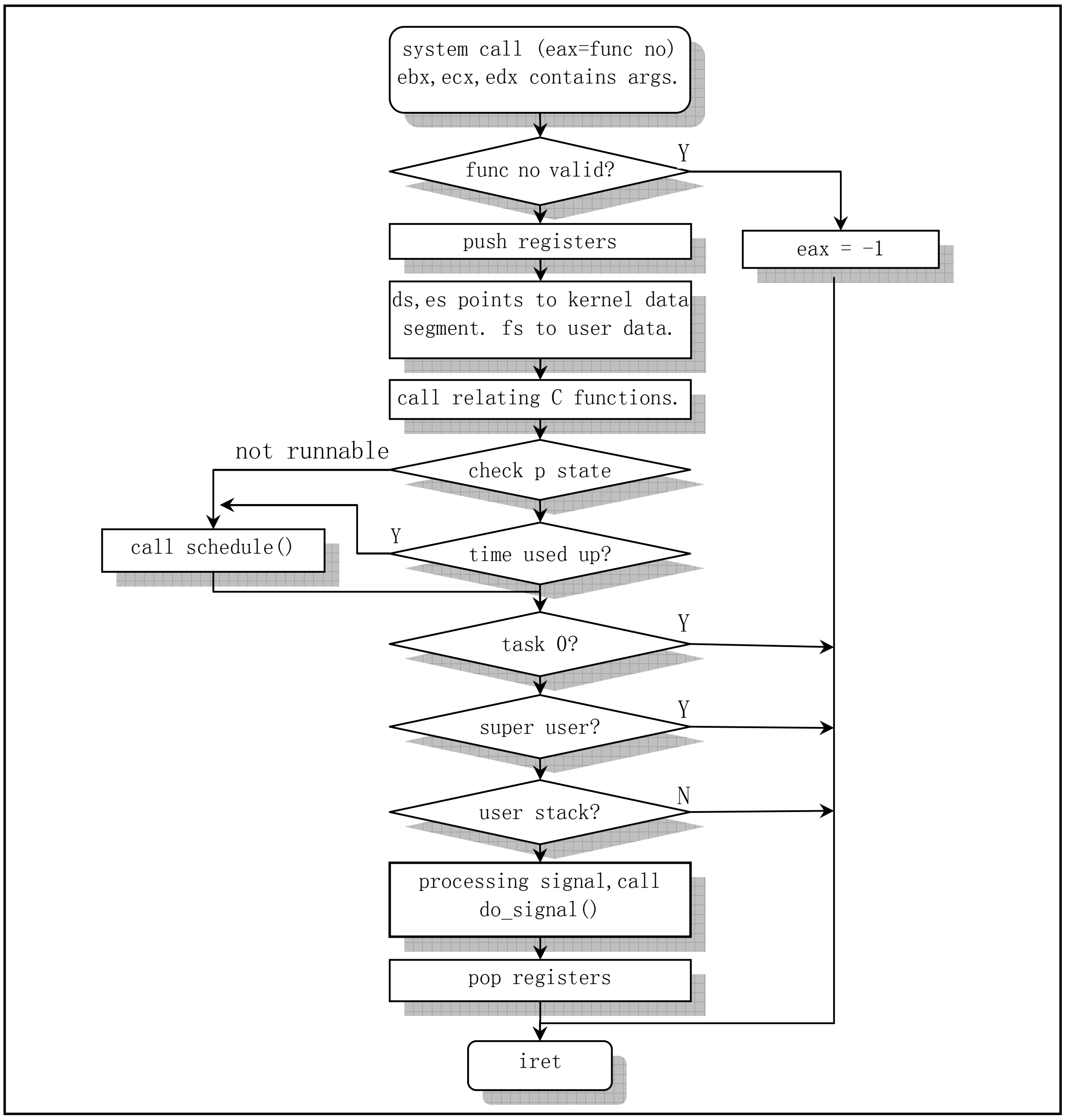
system call (eax=func no)
ebx,ecx,edx contains args.
func no valid?
Y
eax = -1
push registers
ds,es points to kernel data segment. fs to user data.
call relating C functions.
not runnable
check p state
Y
call schedule()
time used up?
Y
task 0?
Y
super user?
N
user stack?
processing signal,call do_signal()
pop registers
iret

Figure 5-5 The flow chart of system call interrupt.

5.6.2 Comments

Program 5-4 linux/kernel/system_call.s

```
/*
 *  linux/kernel/system_call.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  system_call.s  contains the system-call low-level handling routines.
 * This also contains the timer-interrupt handler, as some of the code is
 * the same. The hd- and flopppy-interrupts are also here.
 *
 * NOTE: This code handles signal-recognition, which happens every time
 * after a timer-interrupt and after each system call. Ordinary interrupts
 * don't handle signal-recognition, as that would clutter them up totally
 * unnecessarily.
 *
 * Stack layout in 'ret_from_system_call':
 *
 *       0(%esp) - %eax
 *       4(%esp) - %ebx
 *       8(%esp) - %ecx
 *       C(%esp) - %edx
 *      10(%esp) - %fs
 *      14(%esp) - %es
 *      18(%esp) - %ds
 *      1C(%esp) - %eip
 *      20(%esp) - %cs
 *      24(%esp) - %eflags
 *      28(%esp) - %oldesp
 *      2C(%esp) - %oldss
 */

SIG_CHLD        = 17          # Number of SIG_CHLD signal. (child stopped or end)

EAX             = 0x00        # The relative offset of registers on stack.
EBX             = 0x04
ECX             = 0x08
EDX             = 0x0C
FS              = 0x10
ES              = 0x14
DS              = 0x18
EIP             = 0x1C
CS              = 0x20
EFLAGS          = 0x24
OLDESP          = 0x28        # When privilege changes.
OLDSS           = 0x2C

# The relative offset of some fields in task_struct. Refer to include/linux/sched.h.
state   = 0             # these are offsets into the task-struct.
```

```
49 counter = 4             # ticks of task running (Time slice), decreasing.
50 priority = 8            # counter=priority when start running.
51 signal  = 12            # signal map, each bit associated with one kind of signal.
52 sigaction = 16          # MUST be 16 (=len of sigaction). The offset of sigaction struct.
53 blocked = (33*16)       # signal blocked.
54
55 # offsets within sigaction          # Refer to include/signal.h
56 sa_handler = 0                      # signal handler.
57 sa_mask = 4                         # siganl mask.
58 sa_flags = 8                        # signal flags.
59 sa_restorer = 12                    # Refer to kernel/signal.c.
60
61 nr_system_calls = 72   # Total system call number in Linux 0.11
62
63 /*
64  * Ok, I get parallel printer interrupts while using the floppy for some
65  * strange reason. Urgel. Now I just ignore them.
66  */
67 .globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
68 .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
69 .globl _device_not_available, _coprocessor_error
70
   # Wrong system call number (out of range).
71 .align 2                            # align to 4 bytes bound.
72 bad_sys_call:
73         movl $-1,%eax               # set eax = -1
74         iret
   # Do scheduling. Refer to L104 in kernel/sched.c
75 .align 2
76 reschedule:
77         pushl $ret_from_sys_call    # Push the return address (L101).
78         jmp _schedule
   #### int 0x80 -- Entry point for Linux system call (int 0x80, eax = function number).
79 .align 2
80 _system_call:
81         cmpl $nr_system_calls-1,%eax   # Check the function number in eax.
82         ja bad_sys_call
83         push %ds                    # Save registers.
84         push %es
85         push %fs
86         pushl %edx                  # ebx,ecx,edx contains arguments for C function if any.
87         pushl %ecx                  # push %ebx,%ecx,%edx as parameters
88         pushl %ebx                  # to the system call
89         movl $0x10,%edx             # set up ds,es to kernel space
90         mov %dx,%ds
91         mov %dx,%es
92         movl $0x17,%edx             # fs points to local data space
93         mov %dx,%fs
   # The calling address = _sys_call_table + %eax * 4
   # _sys_call_table is an address arry of system call functions.
94         call _sys_call_table(,%eax,4)
95         pushl %eax                  # Push the return value of function.
96         movl _current,%eax          # Get the address of current process --> eax.
```

```
    # The following lines check the state of current process. If it isn't in running state
    # ( state !=0) or it is in running state but its time is used up, then do rescheduling.
97          cmpl $0,state(%eax)         # state
98          jne reschedule
99          cmpl $0,counter(%eax)       # counter
100         je reschedule
    # The following code handling the siganl of current process.
101 ret_from_sys_call:
    # Check to see whether the current process is task 0, if true then return immediately.
    # The '_task' on L103 is the array name of task[] in C. Direct reference to an array name
    # equals refering to task[0].
102         movl _current,%eax          # task[0] cannot have signals
103         cmpl _task,%eax
104         je 3f                       # Jump forward to label 3.
    # Checking the segment selector of the calling process to see whether the process is kernel
    # code. If true then return immediately. If false, then handle the signal of the process.
    # The following code compares the old code segment selector with user code segment selector
    # 0x0f (RPL = 3, LDT, segment 1 (Local code segment)).
105         cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
106         jne 3f
    # If the stack selector is not 0x17 (that is it's not a user stack), then return.
107         cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
108         jne 3f
    # The following code (L109-120) is used to handle process signal. First it gets the signal
    # map from the task_struct (32-bit, one bit for a signal). Then blocks the siganls with the
    # the signal mask. After that it gets the signal with lowest value and reset the siganl in
    # signal map. And, finally, invokes the do_signal() with the siganl as its one argument.
    # do_siganl() is implemented at L82 in kernel/signal.c and has 13 arguments.
109         movl signal(%eax),%ebx      # Get signal map --> ebx.
110         movl blocked(%eax),%ecx     # Get signal block mask --> ecx.
111         notl %ecx                   # Inverse each bit.
112         andl %ebx,%ecx              # Obtain the permissive signal map.
113         bsfl %ecx,%ecx              # Scan the map from bit 0. ecx will contains seq. number.
114         je 3f                       # If no signal then return.
115         btrl %ecx,%ebx              # Reset the signal (ebx contains the orignal map).
116         movl %ebx,signal(%eax)      # Save the siganl map --> current->signal.
117         incl %ecx                   # Adjust the signal value start from 1 (1-32).
118         pushl %ecx                  # Push the siganl value as one of the arguments for
119         call _do_signal             # function do_signal().
120         popl %eax                   # Pop out the siganl value.
121 3:      popl %eax
122         popl %ebx
123         popl %ecx
124         popl %edx
125         pop %fs
126         pop %es
127         pop %ds
128         iret
129
    #### int 16 -- Handling the error of coprocessor.
    # Jump to invoke C function math_error() at L82 in kernel/math/math_emulate.c. When return,
    # it will continue executing start from ret_from_sys_call.
130 .align 2
```

```
_coprocessor_error:
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        movl $0x10,%eax           # ds,es points to kernel space.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax           # fs points to user space.
        mov %ax,%fs
        pushl $ret_from_sys_call  # Push the return address.
        jmp _math_error           # Invoking the C function math_error().

#### int7 -- Device not available (Coprocessor not available).
# The CPU interprets the pattern 11011B in the first five bits of an instruction as an opcode
# intended for a coprocessor. Instructions thus marked are called ESCAPE or ESC instructions.
# If the CR0's EM (Emulation Mode) flag is set, it means that we should emulate the coprocessor
# functions. If the CPU finds EM set when executing an ESC instruction, it signals this
# exception, giving the exception handler an opportunity to emulate the ESC instruction.
# The TS (Task switched) flag of CR0 is set while CPU does task switching. So we can depend
# on this flag to know whether the contents of the copprocess matches with the current task
# and do some updates of the state of the coprocessor, if necessary. Refer to L77 in sched.c.
.align 2
_device_not_available:
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        movl $0x10,%eax           # ds,es points to kernel data space.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax           # fs points to user space.
        mov %ax,%fs
        pushl $ret_from_sys_call  # Push the return address.
        clts                      # clear TS so that we can use math
        movl %cr0,%eax
        testl $0x4,%eax           # EM (math emulation bit)
# If this exception is not caused by EM flag, then update the state of coprocessor for new task.
        je _math_state_restore    # Invoke the function at L77 in kernel/sched.c.
        pushl %ebp
        pushl %esi
        pushl %edi
# Else we should emulate the coprocessor in program kernel/math/math_emulate.c.
        call _math_emulate
        popl %edi
        popl %esi
        popl %ebp
        ret                         # ret(jump) to label ret_from_sys_call.

#### int32 -- (int 0x20) Timer interrupt handler.
# The Linux system timer is set to the frequency of 100Hz (at L5 in include/linux/sched.h)
# by initializing the 8253 timer chip at L406 in kernel/sched.c. Thus the system time unit
# jiffies is 10 milliseconds.
# This exception handler first increases jiffies by 1, sends the EOI command to nofiy
# the end of the interrupt for interrupt controller (PIC), then invokes the C function
# do_timer() with current privilege level (CPL) as its argument.
.align 2
_timer_interrupt:
        push %ds                    # save ds,es and put kernel data space
        push %es                    # into them. %fs is used by _system_call
        push %fs
        pushl %edx                  # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx                  # save those across function calls. %ebx
        pushl %ebx                  # is saved as we use that in ret_sys_call
        pushl %eax
        movl $0x10,%eax             # ds,es points to kernel space.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax             # fs points to user space.
        mov %ax,%fs
        incl _jiffies
# As the kernel does not initialize PIC to use auto EOI, so here we send EOI command manually.
        movb $0x20,%al              # EOI to interrupt controller #1
        outb %al,$0x20              # OCW2 send to port 0x20 of PIC
# The following code is used to get CPL from old cs selector and pushed onto stack as a
# argument for C function do_timer().
        movl CS(%esp),%eax
        andl $3,%eax                # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
# The jobs of do_timer() is to do task switching, time conunting (at L305 in sched.c).
        call _do_timer              # 'do_timer(long CPL)' does everything from
        addl $4,%esp                # task switching to accounting ...
        jmp ret_from_sys_call

#### This is the entry point of system call sys_execve. It invokes the C function
# do_execve() (L182 of fs/exec.c) with the old eip as its argument.
.align 2
_sys_execve:
        lea EIP(%esp),%eax
        pushl %eax                  # Push old eip as an argument of do_execve().
        call _do_execve
        addl $4,%esp                # Get rid of the argument.
        ret

#### Entry point for system call sys_fork() used to fork a new process.
# It first call the C function find_empty_process() to get a new pid number, then invokes
# the C function copy_process() to duplicate the contents of the process for child.
.align 2
_sys_fork:
        call _find_empty_process    # L135 in kernel/fork.c
        testl %eax,%eax
        js 1f
        push %gs                    # These are part of the arguments for copy_process.
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process          # L68 in kernel/fork.c
        addl $20,%esp               # Get rid of the arguments pushed.
1:      ret

#### int 46 -- (int 0x2E) Hard disk interrupt handler responding hardwrae int request IRQ14.
# This interrupt is signaled when the request hard disk operating finished or encounters an
# error during processing. Refer to kernel/blk_drv/hd.c file.
# This handler first send EOI to the slave 8259A chip, then gets and checks the C function
# pointer in variable do_hd. If the pointer is null, it invokes unexpected_hd_interrupt() to
# show the error message, else it will invokes the C function using the pointer in do_hd.
# Register edx is used to store the pointer of C functions, which is one of read_intr(),
# write_intr() or unexpected_hd_interrupt().
_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax             # ds,es points to kernel space.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax             # fs points to user space.
        mov %ax,%fs
# As the kernel does not initialize PIC to use auto EOI, so here we send EOI command manually.
        movb $0x20,%al
        outb %al,$0xA0              # EOI to interrupt controller #1 (slave 8259A chip)
        jmp 1f                      # give port chance to breathe
1:      jmp 1f
# do_hd is defined as a C func pointer. When the pointer in do_hd is stored in edx, it will
# be set to null.
1:      xorl %edx,%edx
        xchgl _do_hd,%edx
# Check to see if do_hd is null orignally. If null, then edx points to unexpected_hd_interrupt()
# implemented at L237 in file kernel/blk_drv/hdc.
        testl %edx,%edx
        jne 1f
        movl $_unexpected_hd_interrupt,%edx
1:      outb %al,$0x20              # Send EOI to master 8259A chip.
        call *%edx                  # "interesting" way of handling intr.
        pop %fs                     # The above calls the C function pointed by edx.
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret

#### int 38 -- (int 0x26) Interrupt handler for floppy drive. Responding int request IRQ6.
# The processing way is similar as for hard disk exception int 46. Refer to blk_drv/floppy.c
# This handler first send EOI to the slave 8259A chip, then gets and checks the C function
# pointer in variable do_floppy. If the pointer is null, it invokes nexpected_floppy_interrupt()
# to show the error message, else it will invokes the C function using the pointer in do_floppy.
# Register eax is used to store the pointer of C functions, which is one of rw_interrupt(),
# seek_interrupt(), recal_interrupt(), reset_interrupt() or unexpected_floppy_interrupt().
_floppy_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax             # ds,es points to kernel space.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax             # fs points to user space.
        mov %ax,%fs
# As the kernel does not initialize PIC to use auto EOI, so here we send EOI command manually.
        movb $0x20,%al
        outb %al,$0x20              # EOI to interrupt controller #1 (master 8259A chip)
# do_floppy is defined as a C func pointer. When the pointer in do_hd is stored in edx, it
# will be set to null.
        xorl %eax,%eax
        xchgl _do_floppy,%eax
# Check to see if do_floppy is null orignally. If null, then sets register edx to point to
# unexpected_floppy_interrupt(),which implemented in file kernel/blk_drv/floppy.c.
        testl %eax,%eax
        jne 1f
        movl $_unexpected_floppy_interrupt,%eax
1:      call *%eax                  # "interesting" way of handling intr.
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret

#### int 39 -- (int 0x27) Interrupt handler for parallel port. Responding request IRQ7.
# Not yet implemented in this kernel.
_parallel_interrupt:
        pushl %eax
        movb $0x20,%al
        outb %al,$0x20
        popl %eax
        iret
```

5.6.3 Infomation

5.6.3.1 The representation of addressing methods in GNU assembly

The GNU as or inline assembly in gcc uses AT&T syntax. The formal 32 bits addressing format is as follows:

```
AT&T: immed32(basepointer, indexpointer, indexscale)
Intel: [basepointer + indexpointer*indexscal + immed32]
```

The calculation method of this format is : immed32 + basepointer + indexpointer * indexscale. In an application we needn't write out all these fields. But one of the immed32 and basepointer must exists. The following are some examples.

◆ Addressing a particular C variable:

```
AT&T: _booga                        Intel: [_booga]
```

Notice that the underscore ('_') is how you get at static C variables from assembler. This only works with global variables. Otherwise, you can use extended asm to have variables preloaded into registers for you.

◆ Addressing what a register points to:

```
AT&T: (%eax)                        Intel: [eax]
```

◆ Addressing a variable offset by a value in a registr:

```
AT&T: _variable(%eax)               Intel: [eax + _variable]
```

◆ Addressing a value in an array of integers (scaling up by 4):

```
AT&T: _array(,%eax,4)               Intel: [eax*4 + _array]
```

◆ You can also do offsets with the immediate value:

```
C code: *(p+1) where p is a char *
AT&T:  1(%eax) where eax has the value of p      Intel: [eax + 1]
```

◆ You can do some simple math on the immediate value:

```
AT&T: _struct_pointer + 8           Intel: [_struct_pointer + 8]
```

◆ Addressing a particular char in an array of 8-character records. eax holds the number of the record desired. ebx has the wanted char's offset within the record.

```
AT&T: _array(%ebx,%eax,8)           Intel: [ebx + eax*8 + _array]
```

5.6.3.2 Method for adding new system call

To add a new system call to the Linux 0.11 kernel, we need to do the following things.

First we should write a C function processing the system call, for example, sys_sethostname(). This function is used to change the hostname of the system. Usually, we can put this function in file kernel/sys.c. You can certainly put your system call function in a proper location in the kernel source tree. The following program list is one kind of implementation of sys_sethostname(), selected from Linux kernel 0.12, as an example here.

Notice that this function uses variable thisname of structure utsname type in it. This variable is defined in function sys_uname() internally. So we need to move it out from sys_uname() (at L218-220 in sys.c) to be as a global difinition. The hostname field of the is set as an constant value "linux .0" in the file.

```
#define MAXHOSTNAMELEN 8
int sys_sethostname(char *name, int len)
{
        int     i;

        if (!suser())
                return -EPERM;
        if (len > MAXHOSTNAMELEN)
                return -EINVAL;
        for (i=0; i < len; i++) {
                if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
                        break;
        }
        if (thisname.nodename[i]) {
                thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
        }
        return 0;
}
```

Then, we need to allocate a number to this new system call in the header file include/unistd.h. Since Linux 0.11 has 72 system calls (number from 0-71), we append this new one at the end of it. So, we can add a macro difinition for it as follows (properly inserted after line 131). We also need to add the new system call function prototype in the same header file .

```
// In include/unistd.h file.
// New system call number. properly inserted after line 131.
#define __NR_sethostname    72

// The function prototype. properly inserted just after line 251.
int sethostname(char *name, int len);
```

After that, we need to modify the function address table sys_call_table[] of the system calls to append the new function address. This array type table is located in file include/linux/sys.h. The modification is shown as following. Notice that the system call function pointers must be arranged in sequence in the array according to their function number.

```
// In file include/linux/sys.h

extern int sys_sethostname();

// An array table of function pointers for system calls. The new function name is appended at
// the end of the array.
fn_ptr sys_call_table[] = { sys_setup, sys_exit,sys_fork, sys_read,
 ...,
sys_setreuid, sys_setregid, sys_sethostname };
```

Now, we should tell the kernel that the total number of system calls has increased. This can be achived by increasing the number of nr_system_calls by one at L61 in file system_call.s.

At the end, we can recompile to build a kernel Image file which contains our new system call function. To test this new function in our application programs, we may add a interface program in our C library as follows. Refer to chapter 12 for a description of a similar program.

```
#define __LIBRARY__
#include <unistd.h>

_syscall2(int, sethostname, char *, name, int, len);
```

5.7 mktime.c

5.7.1 Function

This program continas one function mktime(), which is used only in init/main.c file for kernel initialization. It calculates the time passed in seconds beginning from the first day in 1970 to the moment of kernel power on.

5.7.2 Comments

Program 5-5 linux/kernel/mktime.c

```
/*
 *  linux/kernel/mktime.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <time.h>

/*
 * This isn't the library routine, it is only used in the kernel.
 * as such, we don't care about years<1970 etc, but assume everything
 * is ok. Similarly, TZ etc is happily ignored. We just do everything
 * as easily as possible. Let's find something public for the library
 * routines (although I think minix times is public).
 */
/*
 * PS. I hate whoever though up the year 1970 - couldn't they have gotten
 * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
 */
#define MINUTE 60                     // Seconds in a minute.
#define HOUR (60*MINUTE)              // Seconds in an hour.
#define DAY (24*HOUR)                 // Seconds in a day.
#define YEAR (365*DAY)                // Seconds in a year.

/* interestingly, we assume leap-years */
// Bellow is an array of time begin from the first day of a year to the start of each month.
static int month[12] = {
        0,
        DAY*(31),
        DAY*(31+29),
        DAY*(31+29+31),
        DAY*(31+29+31+30),
        DAY*(31+29+31+30+31),
        DAY*(31+29+31+30+31+30),
        DAY*(31+29+31+30+31+30+31),
        DAY*(31+29+31+30+31+30+31+31),
        DAY*(31+29+31+30+31+30+31+31+30),
        DAY*(31+29+31+30+31+30+31+31+30+31),
        DAY*(31+29+31+30+31+30+31+31+30+31+30)
};

// Calculate the time between January 1 1970 and system power on in seconds.
// The argument tm is a structure, each field of it is filled with values collected from CMOS.
long kernel_mktime(struct tm * tm)
{
        long res;
        int year;

        year = tm->tm_year - 70;            // Years from 1970 to now. suck 2k problem.
/* magic offsets (y+1) needed to get leapyears right.*/
// Seconds for these years plus one more day for each leapyear and month in current year.
        res = YEAR*year + DAY*((year+1)/4);
        res += month[tm->tm_mon];
/* and (y+2) here. If it wasn't a leap-year, we have to adjust */
        if (tm->tm_mon>1 && ((year+2)%4))
                res -= DAY;
        res += DAY*(tm->tm_mday-1);         // Plus seconds of days in current month.
        res += HOUR*tm->tm_hour;            // Plus seconds of hours in the current day.
        res += MINUTE*tm->tm_min;           // Plus seconds of minutes in the current hour.
        res += tm->tm_sec;                  // Plus seconds in the current minute.
        return res;                         // Finally get the seconds passed from 1970.1.1
}

```

5.7.3 Information

5.7.3.1 Leapear calculation method

The year is a leapyear if it can be divided exactly by 4 and cannot be divided exactly by 100, or can be divided exactly by 400.

5.8 sched.c

5.8.1 Function

Program sched.c is mainly concerned with task switching management in the kernel. The primary functions include sleep_on(), wakeup(), schedule() and do_timer(). In addition, it also contains several C functions for system calls and other "trivial" routines. To facilitate the kernel programming, Linus put the timing routines about floppy drives here too.

schedule() function first checks all tasks to activate every task that has received signal(s). For each task in the kernel, it checks alarm time value in the task's structure. If the alarm time has expired (alarm<jiffies), it set the SIGALRM signal in the signal map of the task and then reset the alarm value to zero. After ignoring the signals blocked, if the task still has other signal and its running state is in TASK_INTERRUPTIBLE, then schedule() will change its state to TASK_RUNNING.

After that, the schedule() function enters into task switching part. It selects the task to be run based on the time slice and the priority scheduling mechanism. First, it checks the counter value of all tasks in ready state. The counter value represents the remaining running time before the task be switched. It selects the task that has the largest value among all tasks and switches to it by using the switch_to() function. If the counter value of all ready to run tasks are zero, which means their running time were used out, then the scheduler will reset the counter value of all tasks, including the tasks at other states, based on their priority value. Finally, it repeat the processing above to selcet one task to run.

Other two functions need to be mentioned are sleep_on() and wake_up(). sleep_on() is used by a process to enter into sleep state by itself. wake_up() is used to change the state of a task to TASK_RUNNING manually. Although small in size, these two functions are somehow hard to understand. Here we discuss them with the help of graph illustrating.

The main function of sleep_on() is to put the current process into sleep state (TASK_UNINTERRUPTIBLE) in a waiting queue for a period of time when the resources it requests is busy or not in memory. A task in TASK_UNINTERRUPTIBLE state need to be wake up manually by using the function wake_up(). The way the task be put into a waiting queue is to use the temporary pointer 'tmp' in the function to be as the relationship links between each waiting tasks in the queue.

The sleep_on() function manipulates three task pointers: '*p', 'tmp' and 'current'. '*p' is a argument of the function indicating the header pointer of the waiting queue. It represents, for eaxmple, the i_wait pointer in the file system waiting queue, the buffer_wait pointer in buffer operating waiting queue. 'tmp' is used as a link pointer in a waiting queue. 'current' the the pointer of current task. We can illustrated the variation of these pointers in memory in Figure 5-6. The long line in the figure represents the bytes sequence in memory.

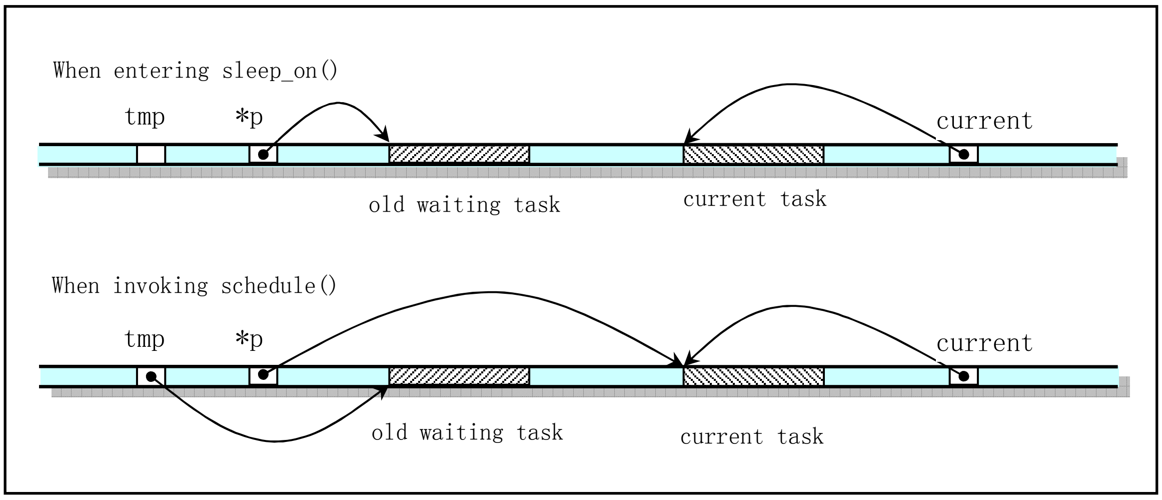

Figure 5-6 The variation of pointer in memory

When just entering the sleep_on() function, the queue header pointer '*p' points to the task already waiting on the queue, or is null if the queue is empty at the beginning. By manipulating these pointers, before invoking the scheduler, the queue header pointer '*p' pointed to the current process while the 'tmp' pointed to the task already waiting in the queue. In other words, the new task (current) is inserted into the queue formed by the links of 'tmp'. Thus, by using the temporary pointer 'tmp' in the function for each waiting task in the queue, a implicit waiting queue is formed. From the Figure 5-7, we can better understand the constructing process of the queue. The figure shows the case when inserting the third task into the waiting queue.

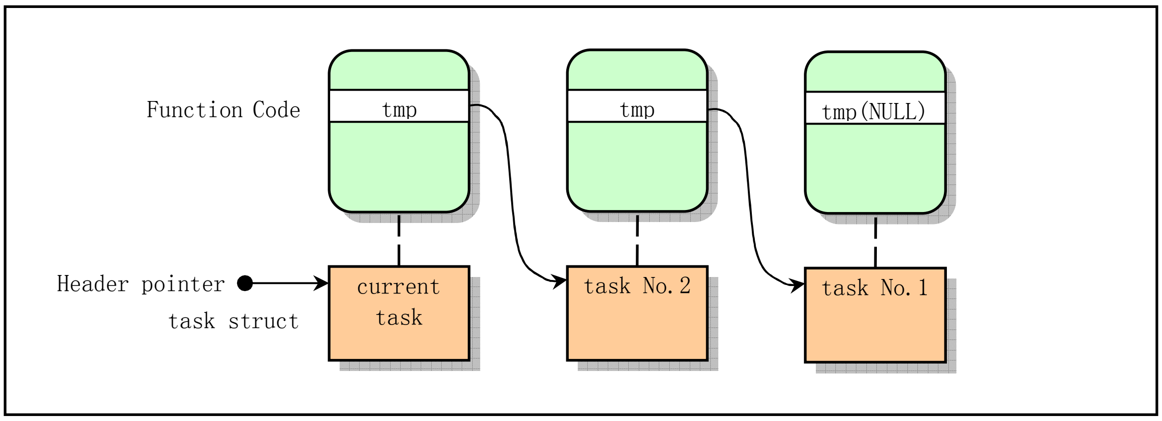

Figure 5-7 The implicit waiting queue for sleep_on()

After the current process was inserted into the queue, sleep_on() invokes the scheduler to switch to run another task. When the task pointed by queue header pointer is waken up, it continues to execute the code bellow 'schedule()' in sleep_on() and wakes up all the other tasks waited in the queue.

The wake_up() function is used to wakes up the tasks waiting for resources on a waiting queue. It is a general waking up function. In some situations, for example to read the data block on a disk, since any task on the waiting queue can be waked up first, the pointer to the waked up task should be set to null. Thus, when the task lately entered into the queue is waked up and continues to run the code in the sleep_on(), it needn't to wake the task with null pointer.

The other function, interruptible_sleep_on(), has the similar structure of sleep_on(). The only difference between them is that the former set the state of the sleeping task into TASK_INTERRUPTIBLE and checks whether there are any other tasks on the queue. In the Linux kenrel 0.12, These two functions are merged into one subroutine and uses the indicated task state as its argument to differentiate these two cases.

When reading the following code, its better to refer to the comments in include/kernel/sched.h file.

5.8.2 Comments

Program 5-6 linux/kernel/sched.c

```
/*
```

```
 *  linux/kernel/sched.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'sched.c' is the main kernel file. It contains scheduling primitives
 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 * call functions (type getpid(), which just extracts a field from
 * current-task
 */
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/sys.h>
#include <linux/fdreg.h>
#include <asm/system.h>
#include <asm/io.h>
#include <asm/segment.h>

#include <signal.h>

// Get the bit mask (binary value) corresponding to signal nr in signal bitmap. The signal
// value range is 1 to 32. e.g.: nr=5, then the bit mask = 1<<(5-1) = 16 = 00010000b.
#define _S(nr) (1<<((nr)-1))
// All signals are blockable except signal SIGKILL and SIGSTOP.
#define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))

// Show the task number, pid, state and the kernel stack free space of the nr-th task in
// the task array.
void show_task(int nr,struct task_struct * p)
{
        int i,j = 4096-sizeof(struct task_struct);

        printk("%d: pid=%d, state=%d, ",nr,p->pid,p->state);
// The task's kernel stack is located after the task's struct and on the same page. The loop
// bellow detects the zero value bytes after the task's struct on the page.
        i=0;
        while (i<j && !((char *)(p+1))[i])
                i++;
        printk("%d (of %d) chars free in kernel stack\n\r",i,j);
}

// Show the task number, pid, state and the kernel stack free space of all tasks.
void show_stat(void)
{
        int i;

// NR_TASKS (64) is the total task number the system can hold at one time. It's also the
// size of array task[] and is defined at L4 in file include/kernel/sched.h.
        for (i=0;i<NR_TASKS;i++)
                if (task[i])
                        show_task(i,task[i]);
}
```

```
45
   // The input frequency of timer 8253 chip is 1.193180MHz. Linux wishes the timer interrupt
   // is 100 Hz, that is generating an interrupt request signal IRQ0 every 10ms (every tick).
   // Thus the LATCH is the initial value for the channel 0 of the 8253 chip. Refer to L407.
46 #define LATCH (1193180/HZ)
47
48 extern void mem_use(void);          // [??] Not used.
49
50 extern int timer_interrupt(void);   // L176 in file kernel/system_call.s
51 extern int system_call(void);       // L80 in kernel/system_call.s
52
   // Define a task union. The task descriptor and the task kernel stack is on the same
   // memory page. Each task uses its own kernel stack when run in kernel mode. The top of
   // the stack is at the end of the page.
53 union task_union {
54         struct task_struct task;
55         char stack[PAGE_SIZE];
56 };
57
   // Defined the default data of the first task (task 0 or called the idle process).
58 static union task_union init_task = {INIT_TASK,};
59
   // Variable 'jiffies' is the time ticks begin from the power on of the system. The type-
   // qualifier 'volatile' is used to suppress optimization that could otherwise occur to the
   // volatile object (jiffies) by the compiler. Defined at L139 in include/linux/sched.h.
60 long volatile jiffies=0;
61 long startup_time=0;                            // Power on time (secs) from 1970.1.1
62 struct task_struct *current = &(init_task.task); // Pointer points to current task.
63 struct task_struct *last_task_used_math = NULL;
64
65 struct task_struct * task[NR_TASKS] = {&(init_task.task), };
66
   // User stack space. It will be used as a user mode stack for task 0 (the idle process)
   // after the kernel finished initialization. 4 Kb in size.
67 long user_stack [ PAGE_SIZE>>2 ] ;
68
   // Define the pointer of user_stack ss:esp. Refer to L23 in boot/head.s. esp will point
   // the last item of user_stack[].
69 struct {
70         long * a;
71         short b;
72         } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
73 /*
74  *  'math_state_restore()' saves the current math information in the
75  * old math state array, and gets the new ones from the current task
76  */
   // Save the math state array for the switched out task and restore the new ones for the
   // new switched in task.
77 void math_state_restore()
78 {
79         if (last_task_used_math == current) // Return if task does not change.
80                 return;
   // Save the math state if the last task used the math coprocessor. Before issue any other
// math instructions, we need to execute 'fwait' first.
        __asm__("fwait");
        if (last_task_used_math) {
                __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
        }
// Memorize the current task which is to use the math coprocessor. If the current task
// used math before then restore its math state, else it means this is the first time
// the current task is going to use math, then issue initializing command to coprocessor
// and set the 'used_math' flag in the task descriptor.
        last_task_used_math=current;
        if (current->used_math) {           // If used before, then resotore its state.
                __asm__("frstor %0"::"m" (current->tss.i387));
        } else {                            // Else the firt time to use.
                __asm__("fninit"::);        // Issue the initilizing command.
                current->used_math=1;       // Set use math flag.
        }
}

/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;          // A pointer to the pointer of task descriptor.

/* check alarm, wake up any interruptible tasks that have got a signal */

// Check 'alarm' begin from the last task in the task pointer array.
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
// If the task set the alarm time before and now is expired, then issue the SIGALRM signal
// to the task and clear the value of 'alarm'. SIGLARM's default opearation is to terminate
// the task.
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
                                        (*p)->signal |= (1<<(SIGALRM-1));
                                        (*p)->alarm = 0;
                                }
// If this task has got signal(s) that can not be blocked (SIGKILL and SIGSTOP) or not blocked,
// and the task is in state TASK_INTERRUPTIBLE, then change the state of the task to ready.
// Code '~(_BLOCKABLE & (*p)->blocked)' is used to ignore signals blocked. But the signal
// SIGKILL and SIGSTOP cannot be blocked or ignored.
                        if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                        (*p)->state==TASK_INTERRUPTIBLE)
                                (*p)->state=TASK_RUNNING;
                }
```

```
121
122 /* this is the scheduler proper: */
123
124         while (1) {
125                 c = -1;
126                 next = 0;
    // The following code is also doing the loop begin from the last item of the task pointer
    // array. It ignores the null pointers and compares the running time counter value of each
    // task in ready state. The counter value is the remain time slice a task need to run before
    // being switched. The pointer 'next' will point to the task which has the largest counter
    // value and 'c' will contain the counter value.
127                 i = NR_TASKS;
128                 p = &task[NR_TASKS];
129                 while (--i) {
130                         if (!*--p)
131                                 continue;
132                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
133                                 c = (*p)->counter, next = i;
134                 }
    // If there is one task its counter value is great than zero, then the program exits the
    // loop starting from L124 and switches to this task at L141.
135                 if (c) break;
    // Otherwise updates each task's counter value based on their priority and repeat loop.
    // The counter value is set to equal to 'counter /2 + priority' with each own values .
136                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
137                         if (*p)
138                                 (*p)->counter = ((*p)->counter >> 1) +
139                                                 (*p)->priority;
140         }
    // Now we switch to task 'next' selected in the loop above. Since the 'next' is initialized
    // to 0, if there is no other tasks could be switched to, the scheduler will run the task 0
    // (idle process) at system idle time. The task 0 invokes system call pause(), which in turn
    // invokes schedule() again.
141         switch_to(next);
142 }
143
    // C function for system call pause(). Change the state of current task to be in waiting
    // state: TASK_INTERRUPTIBLE, and rescheduling.
    // The pause function will cause the invoking process to sleep until a signal is received
    // that either terminates it or causes it to call a signal-catching fuction. This function
    // only returns when a signal was caught and the signal-catching function returned. In this
    // case puase() returns -1, and errno is set to EINTR. So the full functions of pause() are
    // not yet wholly implemented in kernel 0.11 (until version 0.95).
144 int sys_pause(void)
145 {
146         current->state = TASK_INTERRUPTIBLE;
147         schedule();
148         return 0;
149 }
150
    // Due to resources lacking or waiting for other events, the current process initiatively
    // changes itself into sleep state TASK_UNINTERRUPTIBLE and be putted onto a implicit waiting
    // queue by invoking this function. Processes at this state can be back into running state
    // only through the explicit wake up operation using function wake_up() (at L188).
    // The argument '*p' is the header pointer of a waiting queue.
151 void sleep_on(struct task_struct **p)
152 {
153         struct task_struct *tmp;
154 
    // The object pointed by a pointer can be NULL, but a valid pointer itself can not be null.
155         if (!p)
156                 return;
    // if the current process is init task (task 0), then panic (impossible!).
157         if (current == &(init_task.task))
158                 panic("task[0] trying to sleep");
    // Set the temporary pointer 'tmp' points to the task already waited on the queue (if any) and
    // set the header pointer of queue points to the current task. Variable 'tmp' is created on
    // the task's kernel stack and used as implicit links for the waiting queue.
159         tmp = *p;
160         *p = current;
    // Change the state of current task to sleep and do rescheduling.
161         current->state = TASK_UNINTERRUPTIBLE;
162         schedule();
    // Only when the task is changed back to running state again by explicitly invoking the
    // wake_up() function, will the following code be continued. At this moment, the task is
    // explicitly waked up. While the tasks on the queue are waiting for the same resources or
    // events to occur, they will be waked up together by the code bellow.
163         if (tmp)                            // Wake up the previous task on the queue.
164                 tmp->state=0;
165 }
166 
    // Set the current task into interruptible sleep state, and insert into a waiting queue pointed
    // by pointer '*p'. The processing way of this function is similar as function sleep_on().
167 void interruptible_sleep_on(struct task_struct **p)
168 {
169         struct task_struct *tmp;
170 
171         if (!p)
172                 return;
173         if (current == &(init_task.task))
174                 panic("task[0] trying to sleep");
175         tmp=*p;
176         *p=current;
    // Change the state of current task to sleep and do rescheduling.
177 repeat: current->state = TASK_INTERRUPTIBLE;
178         schedule();
    // If the queue has other waiting tasks and the header pointer of the queue doest not point
    // to the current task, then change the state of the task pointed by the header pointer to
    // running state and let the current task sleep again.
    // When the '*p' does not point to the current task, it means there is at least one new task
    // be inserted into the waiting queue after the current task. Thus we should wake up these
    // tasks too. This also indicates that the last task inserted in the queue will be the first
    // one to use the resources waited for.
179         if (*p && *p != current) {
180                 (**p).state=0;
181                 goto repeat;
182         }
    // The following line should be '*p = tmp', that is let the header pointer to point to the
    // remain tasks on the queue. Otherwise, the tasks previously inserted on the queue are
    // erroneously ignored. In addition, the code on L192 should be deleted at the same time.
183         *p=NULL;
184         if (tmp)
185                 tmp->state=0;
186 }
187
    // Waking up the task pointed by '**p'.
188 void wake_up(struct task_struct **p)
189 {
190         if (p && *p) {
191                 (**p).state=0;   // Change the task to running state.
192                 *p=NULL;
193         }
194 }
195
196 /*
197  * OK, here are some floppy things that shouldn't be in the kernel
198  * proper. They are here because the floppy needs a timer, and this
199  * was the easiest way of doing it.
200  */
    // The following code (L201-262) deals with the timing operation of floppy drives. It is
    // recommended to read this block of code with program floppy.c in the next chapter simultaneously
    // or after. The time unit tick equals 1/100 second.

    // The array contains the task descriptor pointer waited for the floppy motor speed up.
    // Array item 0 - 3 corresponding to the floppy drive 0 - 3 (A - D).
201 static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
    // Before data can be transferred to or from the diskette, the drive motor must be brought
    // up to speed. The bellow array will filled with the motor speed up delay time (in ticks)
    // for each floppy drives. For most floppy drives, the spin-up time is 50 ticks (0.5 second).
202 static int  mon_timer[4]={0,0,0,0};
    // The bellow array hold the remainning spin time for each floppy drive before stopped.
    // The default time is 10000 ticks (100 seconds).
203 static int moff_timer[4]={0,0,0,0};
    // Digital Output Register (DOR) contains the drive select and motor enable bits, a reset bit
    // and a DMAGATE# bit. This variable holds the current value of DOR sent to de floppy controller.
    // Bit 7-4: Each bit controls the motor for drive D-A seperately. 1 - start; 0 - stop.
    // Bit 3   : 1 - Enable DMA and interrupt request; 0 - Disable DMA and interrupt request.
    // Bit 2   : 1 - Enable floppy controller; 0 - Disable floppy controller.
    // Bit 1-0: 00 -11 selects drive A to D.
204 unsigned char current_DOR = 0x0C;         // Enable DMA and int, enable controller.
205
    // Decide the spin-up (motor on) delay time.
    // The argument nr (0 - 3) indicates the diskette drive.
    // Returns the spin-up delay time in ticks.
206 int ticks_to_floppy_on(unsigned int nr)
207 {
208         extern unsigned char selected;    // Drive selected flag. (L122,blk_drv/floppy.c)
209         unsigned char mask = 0x10 << nr;  // Corresponds to the motor bits in DOR.
210                                           // The high 4-bit of mask is the motor on flag.
```

```
        if (nr>3)
                panic("floppy_on: nr>3"); // Contains no more than 4 floppy drives.
// First we preset the motor turn off delay time (100 seconds). Then merge the contents
// of current_DOR into variable mask.
        moff_timer[nr]=10000;           /* 100 s = very big :-) */
        cli();                          /* use floppy_off to turn it off */
        mask |= current_DOR;
// If there is no selected drive at this moment, then we select the indicated one (nr).
        if (!selected) {
                mask &= 0xFC;
                mask |= nr;
        }
// If the current value of DOR is different with that of the requested, then we output the
// new value in 'mask' to the floppy controller. Here, if the drive motor has not been turned
// on, then we should set the drive's motor spin-up time to 0.5 s (HZ/2 = 50 ticks), else
// we reset it to 2 ticks to satisfy the requests for decreasing first in function
// do_floppy_timer(). Finally, we save the current DOR into variable 'current_DOR'.
        if (mask != current_DOR) {
                outb(mask,FD_DOR);
                if ((mask ^ current_DOR) & 0xf0)
                        mon_timer[nr] = HZ/2;
                else if (mon_timer[nr] < 2)
                        mon_timer[nr] = 2;
                current_DOR = mask;
        }
        sti();
        return mon_timer[nr];         // Return the motor spin-up delay time.
}

// Waiting for the motor to spin-up,then return.
// Argument 'nr' specifies the floppy drive.
// Set the motor spin-up time if the DOR is different with that of the request. Sleep to
// wait for the motor to speed up. In the handle of timing interrupt (L331 in do_timer()),
// the spin-up time (mon_timer[nr]) will be decreased and checked, until the time is up.
void floppy_on(unsigned int nr)
{
        cli();                          // disable interrupt.
// While the motor spin-up time is not over, we put the current task into motor waiting
// queue with uninterruptible state.
        while (ticks_to_floppy_on(nr))
                sleep_on(nr+wait_motor);
        sti();                          // Enable interrupt.
}

// Set the motor spin-down (turn off) delay time for the specified floppy drive.
// If we do not explicitly turn off the motor within 3 seconds using this function, the
// motor will be turned off after 100 seconds.
void floppy_off(unsigned int nr)
{
        moff_timer[nr]=3*HZ;
}

// Timer function for floppy drives.
// Updating the value of motor spin-up and turn off delay time in mon_timer[] and moff_timer[]
// seperately. This function will be invoked in the C function do_timer() of system timing
// interrupt. Thus the kernel invokes the function for each ticks to update the delay time.
// If the delay time for turning off the motor is up, the corresponding motor flag bit in
// DOR will be cleared.
void do_floppy_timer(void)
{
        int i;
        unsigned char mask = 0x10;

        for (i=0 ; i<4 ; i++,mask <<= 1) {
                if (!(mask & current_DOR))             // Skip if not the specified motor.
                        continue;
                if (mon_timer[i]) {
// If the motor spin-up time is over then wakes up the corresponding waiting task(s).
                        if (!--mon_timer[i])
                                wake_up(i+wait_motor);
                } else if (!moff_timer[i]) {
// If the motor spin-down delay time is over, then reset the corresponding motor flag in
// DOR, save and update the DOR in floppy controller.
                        current_DOR &= ~mask;
                        outb(current_DOR,FD_DOR);
                } else
                        moff_timer[i]--;            // Decreasing the turn-off delay time.
        }
}

#define TIME_REQUESTS 64             // Maximum timers.

// 定时器链表结构和定时器数组。
static struct timer_list {
        long jiffies;                // Time to delay (ticks).
        void (*fn)();                // Handler, the address of a function to be called.
        struct timer_list * next;    //
} timer_list[TIME_REQUESTS], * next_timer = NULL; // The header pointer of the timer_list.

// Add timer function.
// The floppy driver (floppy.c) uses this function to control the state of its motor.
// Arguments: jiffies - ticks to delay; *fn() - the handler when timer expired.
void add_timer(long jiffies, void (*fn)(void))
{
        struct timer_list * p;

// The timer handler can not be null.
        if (!fn)
                return;
        cli();
// If the timer value jiffies <=0, then we should process the timer handler right away and
// needn't insert it into the timer list.
        if (jiffies <= 0)
                (fn)();
        else {
// Otherwise, we will add it to the timer list. First search to get a empty item in the
// array timer_list[].If there is no empty item availabe, then the kernel goes panic.
                for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
                        if (!p->fn)
                                break;
                if (p >= timer_list + TIME_REQUESTS)
                        panic("No more time requests free");
// Insert the request timer into and to be the first item of the timer list. The timer
// header pointer points to it.
                p->fn = fn;
                p->jiffies = jiffies;
                p->next = next_timer;
                next_timer = p;
// Now we sort the items of the timer list, the order depends on the delay value, from
// short to long dely time. During the sorting, the time value of a item need to minus the
// sum time of all previous items. Thus, we need only to check the first item in the timer
// list to determine if there is a timer expired.
                while (p->next && p->next->jiffies < p->jiffies) {
                        p->jiffies -= p->next->jiffies;
                        fn = p->fn;
                        p->fn = p->next->fn;
                        p->next->fn = fn;
                        jiffies = p->jiffies;
                        p->jiffies = p->next->jiffies;
                        p->next->jiffies = jiffies;
                        p = p->next;
                }
        }
        sti();
}

// The C function of the handler of system timer interrupt invoked at L176 in system_call.s
// Argument 'cpl' is the current privilege level. If it is zero, means CPU is executing the
// kernel code. This function updates the timer and counting the time used by the current
// process. When the process uses out its time slice and is running its user's code, then
// the CPU will do switching
void do_timer(long cpl)
{
        extern int beepcount;      // beep count of the speaker. (L697 in chr_drv/console.c)
        extern void sysbeepstop(void);  // Stop the speaker. (L691 in chr_drv/console.c)

// If the beep count is reached, then stop the speaker. The way is to send command to port 0x61,
// reset bit 0 and bit 1. Bit 0 controls the channel 2 operation of the 8253 timer chip.
// Bit 1 controls the speaker.
        if (beepcount)
                if (!--beepcount)
                        sysbeepstop();

// Updating the resource usage. If the argument 'cpl' is 0, means the CPU is executing kernel
// code, then increasing the kernel counting time stime of the process, else increasing the
// utime of the process. [By reading the kernel code, we can see that Linus named the kernel
// code as supervisor's code. Refer to Linus' comments at L193 in file system_call.s ]
        if (cpl)
                current->utime++;
```

```
316         else
317                 current->stime++;
318
    // By checking the timer list header pointer, we can decide if there is timer exist in the
    // timer list. If there is one, then minus the time value by 1 to the first timer. If the
    // value equals zero, means the timer is expired, then we invoke its handler and finally
    // remove the item from the timer list.
319         if (next_timer) {                     // timer list header (see L270).
320                 next_timer->jiffies--;
321                 while (next_timer && next_timer->jiffies <= 0) {
322                         void (*fn)(void);     // A definition of a func pointer ! :(
323
324                         fn = next_timer->fn;
325                         next_timer->fn = NULL;
326                         next_timer = next_timer->next;
327                         (fn)();               // invoking the timer handler.
328                 }
329         }
    // If the digital output register (DOR) of the floppy controller contains the motor turning
    // on bit(s), then invokes the floppy timer handler 'do_floppy_timer()' (L245).
330         if (current_DOR & 0xf0)
331                 do_floppy_timer();
    // If the time slice of the current process is not yet expired, then return. Otherwise, if
    // the CPU is executing kernel code before entered into this interrupt handler, we also
    // return right away. This means that the Linux kernel code cannot be preempted. Otherwise
    // we invoke the scheduler to switch the current process.
332         if ((--current->counter)>0) return;
333         current->counter=0;
334         if (!cpl) return;
335         schedule();
336 }
337
    // System call function - Set a timer that will expire at a specified time in the future.
    // When the timer expires, the SIGALRM signal is generted for the process. If we ignore or
    // don't catch this signal, its default action is to terminate the process.
    // Argument 'seconds' is the number of seconds in the future when the signal should be generated.
    // If it greater than 0, then it is set in the alarm field of the process descriptor, else
    // a zero is returned.
338 int sys_alarm(long seconds)
339 {
340         int old = current->alarm;
341
342         if (old)
343                 old = (old - jiffies) / HZ;
344         current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
345         return (old);
346 }
347
    // Get process identification, returns  the process ID of the current process - pid.
348 int sys_getpid(void)
349 {
350         return current->pid;
351 }
```

```

// Returns the process ID of the parent of the current process - ppid.
int sys_getppid(void)
{
        return current->father;
}

// Get user identity, returns the real user ID of the current process - uid.
int sys_getuid(void)
{
        return current->uid;
}

// Returns the effective user ID of the current process - euid.
int sys_geteuid(void)
{
        return current->euid;
}

// Get group identity, returns the real group ID of the current process - gid.
int sys_getgid(void)
{
        return current->gid;
}

// Returns the effective group ID of the current process - egid.
int sys_getegid(void)
{
        return current->egid;
}

// Change process priority, adds 'increment' to the nice value for the current process.
// A large nice value means a low priority. A negative value will increase the privority.
int sys_nice(long increment)
{
        if (current->priority-increment>0)
                current->priority -= increment;
        return 0;
}

// Schedule initialization routine.
void sched_init(void)
{
        int i;
        struct desc_struct * p;    // descriptor structure, refer to include/linux/head.h

        if (sizeof(struct sigaction) != 16)      // sigaction is a signal handling struct.
                panic("Struct sigaction MUST be 16 bytes");
// Setup the init task's TSS and LDT descriptors in the global descriptor table (GDT).
// Refer to lines begin from 65 in file include/asm/system.h
        set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
        set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
// Clear the task pointer array and the descriptors in GDT. Notice that the action begin
// from the second item in the task array, so the init task pointer is still there.
        p = gdt+2+FIRST_TSS_ENTRY;
        for(i=1;i<NR_TASKS;i++) {
                task[i] = NULL;
                p->a=p->b=0;
                p++;
                p->a=p->b=0;
                p++;
        }
/* Clear NT, so that we won't have troubles with that later on */
// The NT (Nested Task) flag in the system EFLAGS controls the chaining of interrupted and
// called tasks. It influences the operation of the IRET instruction. If NT is set, the IRET
// instruction will cause the task switch. Here reset the NT flag.
        __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
// Load the init task's TSS selector into task register TR from the GDT. Load the init
// task's LDT selector into local descriptor table register LDTR from the GDT. Notice that
// the LDTR is only explicitly loaded once at here, all the subsequent loading of LDTR will
// be performed by CPU automatically according to the LDT field in TSS.
        ltr(0);
        lldt(0);
// The following code is used to initialize the channnel 0 of the 8253 timer chip. Select
// mode 3, binary counting method. The output pin of channel 0 is connected to the IRQ0 of
// of the master PIC 8259A chip. It issues a IRQ0 signal every 10 ms. The constant LATCH
// is the initial counting value set to the channel 0, refer to L46.
        outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);    /* LSB */   // Least Significant Byte. (Low Byte)
        outb(LATCH >> 8 , 0x40);        /* MSB */   // Most Significant Byte. (High Byte)
// Install the handler for timer interrupt. Modify the interrupt mask code to enable system
// timer generating interrupts. Install the system call gate (interrup 0x80).
        set_intr_gate(0x20,&timer_interrupt);
        outb(inb_p(0x21)&~0x01,0x21);
        set_system_gate(0x80,&system_call);
}

```

5.9 signal.c

5.9.1 Function

Program signal.c deals with the all functions processing signals. In the UN*X system, signal is one kind of "soft interrupt" mechanism. A few number of complicated applications will use singals. The signal mechanism provides a way to handle asynchronous events. For example, in the case the user enter combine keys 'ctrl-C' on a terminal keyboard to terminate a running program. This operation will cause the kernel to generate signal SIGINT (signal interrupt), which is to be sent to the foreground process and result in the termination of the process in default situation. Another example is when the process has setup a alarm timer and which was expired, the kernel would send a SIGALRM signal to the process. When an hardware exception occurred, the kernel will also send a corresponding signal to the current process too. In addition, a process may send a signal to the other process. For example, a process send a SIGTERM signal using function kill() to a process in the same group to terminate it.

The signal processing mechanism is already existed from the old UN*X system, but the way handling sigals in their kernels is not much reliable. The signal may got lost, and the signal is hard to close in the critical code regin. Afterwards, the POSIX provids a reliable way for handling signals using sigaction structure. To keep compatible with these two handle methods, program signal.c provids them at then same time.

Usually, the kernel uses bits in a unsigned long integer (32-bit) to represente different signals for a process. This integer is called signal map in the process descriptor. Thus, there are at most 32 different signals in the system. In the kernel discussed, 22 different signals were defined which includes all the 20 kinds of signals that defined in the POSIX.1 standard. The other two signals are special ones defined by the Linux kernel. They are SIGUNUSED and SIGSTKFLT (stack fault). The former is used to represent all the other signals the system not yet supported. The later is used when the stack encounter error. The name and the difinition of all the 22 different signals are listed in table at the end of the program. You can also refer to the contents of file include/signal.h for the detailed descriptions.

When a signal is received, the process has three different ways to handle it:

1. Ignore the signal. Most of the signals can be ignored by the process. But the two signals: SIGKILL and SIGSTOP can never be ignored. The reason is that the kernel should provide an affirmatory way for the supervisor to stop or kill any processes. Also, if some of the signals generated by a hardware exception (such as divide-by-zero) are ignored, the behavior of the process is undefined.
2. Catch the signal. To do this, we should tell the kernel the signal handling function before the it occurred. In the signal function we can do whatever we want to handle the condition, or do nothing to ignore the signal. An example is if the process has created some temporary files, we may want to write a signal-catching function for SIGTERM signal to clean up the temporary files.
3. Execute the signal's default action. The kernel provides a default action for all of the signals. The generally action of these default signals is to terminate the process.

This program mainly provides the following signal handling functions:

- sys_ssetmask() - System call to set the signal block code of the current process.
- sys_sgetmask() - System call to get the signal block code of the current process.
- sys_signal()   - System call to handle a signal, i.e. the traditional signal handling function signal().
- sys_sigaction() - System call to to change the action taken by a process on receipt of a specific signal, i.e. the reliable signal handling function sigaction().
- do_signal() - The C function invoked in the system calls interrupt.

The action performed by function signal() and sigaction() is similar. They all used to change the signal handler for a indicated signal. But the way the signal() does may cause a signal lost in some particular situations. The additional two functions related to singal handling are send_sig() and tell_father(), and implemented in program exit.c which is the next program to be commented.

At the line 55 in the header file include/signal.h, the prototype of function signal() is declared as follows:

```
void (*signal(int signr, void (*handler)(int)))(int);
```

The function contains two arguments. The 'signr' is a signal number to catch. The other is a function pointer of the new signal handler which has no return value and contains one integer argument. This integer argument is used by the kernel to send the signal to the signal handler when it occurred.

The signal() installs a new signal handler for the signal with number 'signr'. The signal handler si set to 'handler' which may be a user specified function, or either SIG_IGN or SIG_DFL.

When the specified signal 'signr' arrived, if the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated to the signal performed. Otherwise, if the handler is set to a function 'handler' then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next 'handler' is called with argument 'signr'.

On return, the signal() function returns the previous value of the signal handler which is also a function pointer with no return value and contains one integer argument. After the invocation of the new handler, the handler is reset to the default handler SIG_DFL.

At line 45-46 in file include/signal.h, the default handler SIG_DFL and ignoring handler SIG_IGN is defined as a macro as following:

```
#define SIG_DFL      ((void (*)(int))0)
#define SIG_IGN      ((void (*)(int))1)
```

They all represent function pointers with no return value same as the second agrument in signal(). The pointer value are 0 and 1 seperately. This two value are function addresses that are impossible to happen in the practical function address. Thus in the signal() function, the kernel can check these two value to determine whether to do the default action or ignore the signal for the process. Of cause the SIGKILL and SIGSTOP can not be ignored. Refer to the L94 - 98 in the signal.c program.

When begin running a program, the kernel will set its signals default handler to SIG_DFL or SIG_IGN. In addition, when forking a child process, this child process will inherit all the signals handling method (signal block mask) from its parent. Thus the setup and handling methods set by parent process are also valid in the child process.

To continuely catch a specified signal, function signal() is usually called in a way like the following example:

```
void sig_handler(int signr)         // Signal handler.
{
    signal(SIGINT, sig_handler);    // Reinstall the signal handler to catch the next signal.
    ...
}

main ()
{
    signal(SIGINT, sig_handler);    // Firstly install the signal handler in the main.
    ...
}
```

The reason that the signal() is unreliable is that in the case when a signal was happened and entered into its handler, before we re-installed the signal handler again, there may be a new signal occurred in this short period. But the kernel has set the handler to be the default one, thus may cause a signal to be lost.

The function sigaction() adopts sigaction data structure to save the information of the indicated signal. It is a reliable mechanisim to handling signals. We can easily check information and modify the handler of a specified signal. This function is a super-set of the signal() and is declared at L66 in header file include/signal.h like this:

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
```

This function is used to change the action taken by a process on receipt of a specific signal. The argument 'sig' is the signal number we specified and can be any valid signal except SIGKILL and SIGSTOP. The later two arguments are pointers to the sigaction structure. If 'act' is not null, then the new action for the signal 'sig' is installed from the 'act'. If 'oldact' is non-null, the previous action for the signal is saved in 'oldact'. The sigaction structure is defined as following (in file include/signal.h):

```
48 struct sigaction {
49         void (*sa_handler)(int);        // Signal handler.
50         sigset_t sa_mask;               // Signal mask, used to block specified signal set.
51         int sa_flags;                   // Option flags.
52         void (*sa_restorer)(void);      // Restorer, used internally by the kernel.
53 };
```

'sa_handler' specifies the action to be associated with 'sig' and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. 'sa_mask' gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER or SA_NOMASK flags are used. 'sa_flags' specifies a set of flags which modify the behavior of the signal handling process. It is formed by the bitwise OR of zero or more of the flags described in header file include/signal.h, from L35 to L39.

When changing the handling method for a specified signal, if sa_handler is not one of the default handler SIG_IGN or SIG_DFL, then before the 'sa_handler' is to be invoked, the field 'sa_mask' indicates a signal set to be added into the signal block map of the process. If the singal handler returns, the kernel will restore the signal block map for the process. Thus, we can block a set of specified signals before executing the signal handler. In addition, before executing the signal handler, the modified signal block map also contains the current signal 'sig' to prohibit the sending of current signals. Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we are finished processing the first occurrence. Furthermore, additional occurrences of the same signal are usually not queued. If the signal occurs several times while it is blocked, when we unblock the signal, the signal handling function for that signal will usually be invoked only one time.

Once we install an action for a given signal, that action remains installed until we explicity change it by calling sigaction again. Unlike the function signals(), POSIX.1 requires that a signal handler remain installed until explicitly changed.

The last field of the sigaction structure and the 'restorer' argument in function sys_signal() is a function pointer, which is provided by the library, for instance Libc, at the compilation period. This function is used to do some cleaning process and restore the return value in register eax by the system call. Refer to the following for detailed descriptions.

Function do_signal() is used to preprocess the signals for a task within the system call interrupt int 0x80. When a process invokes a system call, if the process have received signal(s), then this function will insert the signal handler onto the process's user stack. Thus, when exiting from the system call process, the signal handler will be runned right away, after that, the user program will be continued if the signal handler returns. Refer to Figure 5-8.

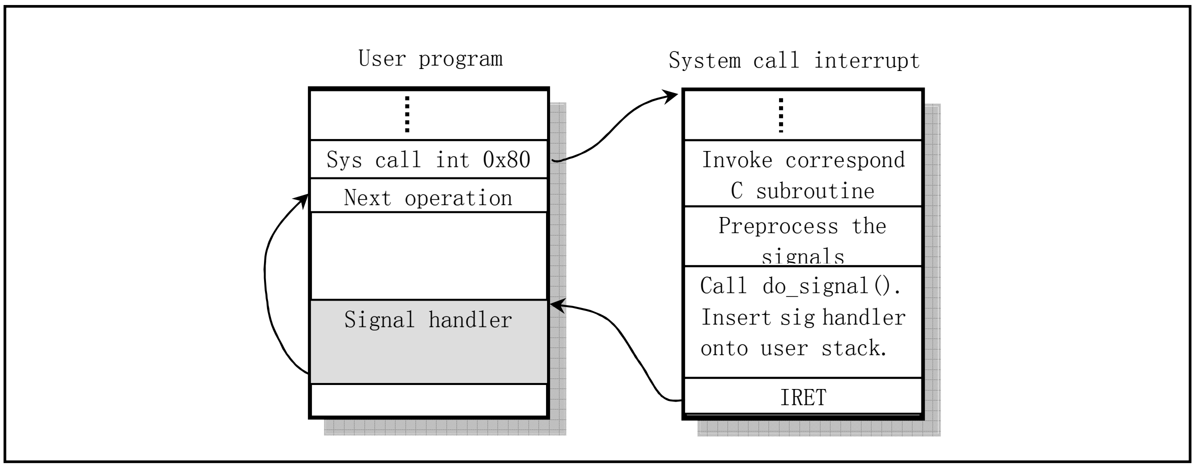

Figure 5-8 The invoking method of the signal handler

Before the insertion of the signal handler onto the user stack, do_signal() will first extends the stack pointer for a few long words ('longs', 7 or 8 long words, see the following program at L106) and, then, filled with arguments in it, as demonstrated in Figure 5-9. Since the code begin from L104 in the following program is hard to understand, here we give them very detailed description.

When the user program (process) invokes system call and begins enter into the kernel code, The kernel stack of the process was filled with some values as shown in Figure 5-9. These values are SS, ESP, EFLAGS and the return address in the user program. When finished handling the corresponding system call function and just before invoking the do_signal() (begin from L118 in program system_call.s), the process's kernel stack contains values as illustrated on the left side in Figure 5-10. This values formed the arguments for the do_signal() function.

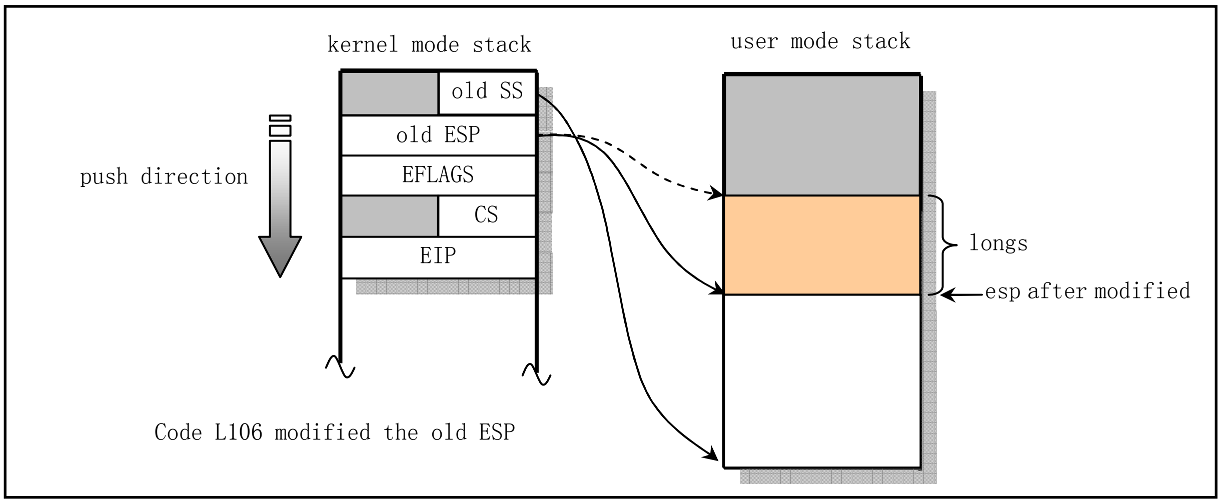

Figure 5-9 User stack modification by function do_signal()

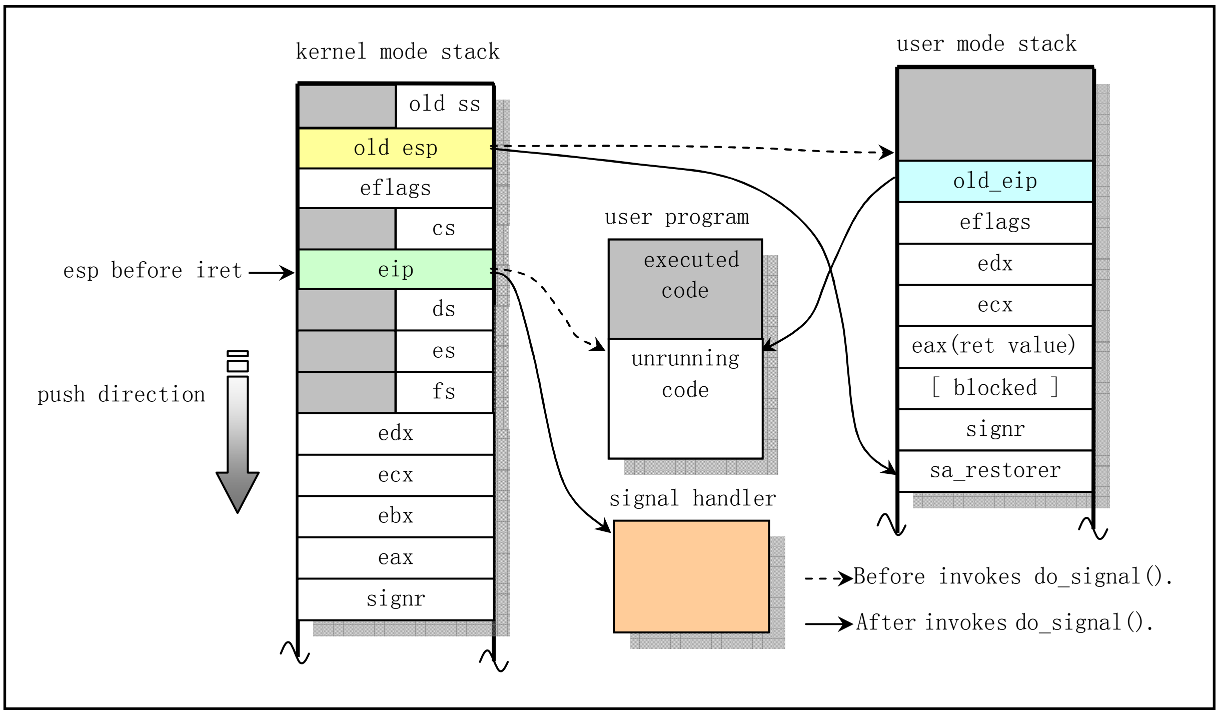

Figure 5-10 The detail modification procedure of user stack by do_signal()

After do_signal() deals with the two default signal handlers (SIG_IGN and SIG_DFL), if the user program have defined its own signal handler ('sa_handler'), then do_signal() start prepareing to insert the user signal handler onto the user mode stack at L104. Firstly, it saves the user program return eip in the kernel mode stack to be as old_eip and replace the eip with the pointer of the user defined signal handler 'sa_handler'. Then, it expands the current using space of the user mode stack by decreasing 'old esp' in 'longs' size. Lastly, it copies a few register values on the kernel mode stack onto the user mode stack as shown in the right side of Figure 5-10.

The total number of parameters putted on the user stack are 7 or 8 long words. We now describe their meanings below.

'old_eip' is the orignal return instruction pointer of the user program. It was saved before the eip be replaced with the user signal handler on the kernel mode stack. 'eflags', 'edx', 'ecx' are the values when user program invokes the system call and are basically the arguments for the system call. When end of the system call, we need to restore these values in the relating registers. 'eax' keeps the system call return code. If the signal allow the reception of itself, then we also need to store the process signal block mask 'blocked' onto the user stack. This is reason why the 'longs' size is one long words variable. The next value is signal number 'signr'. The last one is a signal action restore function pointer 'sa_restorer'. This function is not provided by the user. The user only defines the signal number 'signr' and the signal handler 'handler' for the function signal().

The following is a simple example to setup a signal handler for the signal SIGINT. In the default case, this signal is generated when we press Ctrl-C keys.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
// Signal handler.
void handler(int sig)                       // Signal processing function.
{
    printf("The signal is %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);         // Restore the default signal handler.
}

int main()
{
    (void) signal(SIGINT, handler);         // Install the user defined signal handler.
    while (1) {
        printf("Signal test.\n");
        sleep(1);                           // Wait one second.
    }
}
```

The signal processing function handler() will be invoked when the process receives a SIGINT signal. The handler() first shows a message and then restores the default signal handler for signal SIGINT. Thus when we press the combining keys CTRL-C the second times, this program will be terminated.

Now, lets discuss the function sa_restorer. We cannot find its difinition in the kernel sources, so, where does it come from? As a matter of fact, it was definied in the library file libc.a as following:

```
.globl ____sig_restore
.globl ____masksig_restore
# If there is no 'blocked' then use procedure restorer
____sig_restore:
    addl $4,%esp            # get rid of sinal value 'signr'.
    popl %eax               # restore the return value of the system call.
    popl %ecx               # restore the original register values of the user program.
    popl %edx
    popfl                   # restore the EFLAGS of user program.
    ret
# If there is a 'blocked', then we use the following restorer procedure.
# 'blocked' is used by ssetmask.
____masksig_restore:
    addl $4,%esp            # ignore the signal value 'signr'.
    call ____ssetmask       # Setup signal mask 'old blocking'.
    addl $4,%esp            # discard 'blocked'
    popl %eax
    popl %ecx
    popl %edx
    popfl
    ret
```

The main purpose of the above function is to restore the returnning value, a fews register values and discard the signal value on the stack when finished executing the system calll by the user program at the end of the processing the signal handler. When compiling and linking the user defined signal handling function, the compiler will insert the above function into the user's program and will do the above job for the program as if the kernel return directly from the system call without run the user defined signal handler.

Now we give a sum description of the signal.c program. After finished executing function do_signal(), program system_call.s pops out all values bellow the value of eip on the kernel state stack. After running the instruction 'iret', the CPU pops out cs:eip, eflags and ss:esp and restored to the user state and continue running the user program. Since the eip has been replaced with the signal handler pointer, the CPU executes the user defined signal handler right away. After the executing of signal handler, the CPU passes the control to the function sa_restorer. This function does some clearing and restoring jobs for the user program and return and continue to run the user program.

5.9.2 Comments

Program 5-7 linux/kernel/signal.c

```
/*
 *  linux/kernel/signal.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#include <signal.h>

volatile void do_exit(int error_code);

// Get the mask of blocked signals.
int sys_sgetmask()
```

```
{
        return current->blocked;
}

// Set the new signal blocking mask. SIGKILL can not be blocked. ret the old blocking mask.
int sys_ssetmask(int newmask)
{
        int old=current->blocked;

        current->blocked = newmask & ~(1<<(SIGKILL-1));
        return old;
}

// Copy the data of sigaction to the address 'to' in the user space (fs segment).
static inline void save_old(char * from,char * to)
{
        int i;

// Verify that there is enough memory space at the address 'to' to hold the data.
        verify_area(to, sizeof(struct sigaction));
        for (i=0 ; i< sizeof(struct sigaction) ; i++) {
                put_fs_byte(*from,to);         // Copy to segment fs of user space.
                from++;                        // put_fs_byte() is in include/asm/segment.h
                to++;
        }
}

// Get the sigaction data from user space (fs segment).
static inline void get_new(char * from,char * to)
{
        int i;

        for (i=0 ; i< sizeof(struct sigaction) ; i++)
                *(to++) = get_fs_byte(from++);
}

// The signal() system call. It installs a new signal handler for the signal with number
// 'signum'. The signal handler is set to 'handler' which may be a user specified function
// or either SIG_IGN or SIG_DFL.
// Arguments: signum -- the specified signal;
//            handler -- the signal handler;
//            restorer -- the restoring function pointer.
// The restoring function is provided by library file libc.a. It is used to do the ending
// process for the user program after executing the signal handler.
// This system call returns the orignal signal handler.
int sys_signal(int signum, long handler, long restorer)
{
        struct sigaction tmp;

// the signal number must in the range of 1 to 32, and cannot be signal SIGKILL.
        if (signum<1 || signum>32 || signum==SIGKILL)
                return -1;
// This signal uses the handler only once and then restore to the default handler.
// The signal can received in its handler.
        tmp.sa_handler = (void (*)(int)) handler;     // The specified signal handler.
        tmp.sa_mask = 0;                              // signal mask when running.
        tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
        tmp.sa_restorer = (void (*)(void)) restorer;
        handler = (long) current->sigaction[signum-1].sa_handler;
        current->sigaction[signum-1] = tmp;
        return handler;
}

// The sigaction() system call is used to change the action taken by a process on receipt
// of a specific signal. 'signum' specifies the signal and can be any valid signal except
// SIGKILL (and SIGSTOP). If 'action' is non-null, the new action for signal 'signum' is
// installed from 'action'. If 'oldaction' is non-null, the previous action is saved in it.
// The function returns 0 on success and -1 on error.
int sys_sigaction(int signum, const struct sigaction * action,
        struct sigaction * oldaction)
{
        struct sigaction tmp;

// The valid signal numer is in the range of 1 to 32 and cannot be signal SIGKILL.
        if (signum<1 || signum>32 || signum==SIGKILL)
                return -1;
// Get the user defined sigaction data from 'action' in user space to the current process.
        tmp = current->sigaction[signum-1];
        get_new((char *) action,
                (char *) (signum-1+current->sigaction));
// If 'oldaction' is not null, then save the orignal sigaction data for the user.
        if (oldaction)
                save_old((char *) &tmp,(char *) oldaction);
// If we permit to receive the signal in the signal's handler, then clear the signal mask,
// else set to mask the specified signal 'signum'.
        if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
                current->sigaction[signum-1].sa_mask = 0;
        else
                current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
        return 0;
}

// The signal preprocessing function for the system call interrupt procedure invoked at
// line 119 in kernel/system_call.s file. This function is used to insert the signal
// handler into the user state stack for executing the signal handler right after return
// from the system call.
void do_signal(long signr,long eax, long ebx, long ecx, long edx,
        long fs, long es, long ds,
        long eip, long cs, long eflags,
        unsigned long * esp, long ss)
{
        unsigned long sa_handler;
        long old_eip=eip;
        struct sigaction * sa = current->sigaction + signr - 1; //current->sigaction[signu-1]。
        int longs;
        unsigned long * tmp_esp;

        sa_handler = (unsigned long) sa->sa_handler;
// If the signal handler is SIG_IGN (ignore the signal), then return; if the signal handler
// is SIG_DFL (the default handler), then, if the signal is SIGCHILD then return, else we
// terminate the program.
// The handler SIG_IGN is defined as 1, SIG_DFL is defined as 0. Refer to line 45, 46 in
// file include/signal.h.
        if (sa_handler==1)
                return;
        if (!sa_handler) {
                if (signr==SIGCHLD)
                        return;
                else
                        do_exit(1<<(signr-1));   // This should be do_exit(1<<(signr)).
        }
// If the handler need to run once, then clear the handler for this signal.
        if (sa->sa_flags & SA_ONESHOT)
                sa->sa_handler = NULL;
// The following code is used to insert the signal handler onto user stack, and at the same
// time pushes the restore function pointer 'sa_restorer', signal number 'signr', process
// mask code (if SA_NOMASK isn't set), register eax, ecx, edx, eflags and the user return
// address on to the user stack. Thus, the CPU will run the user specified signal handler
// first and then continue to execute the user program after the system call.

// Change the value of eip on the kernel state stack to the pointer of the signal handler.
        *(&eip) = sa_handler;
// If it allows the signal handler to receive the own signal, then we need to push the
// signal blocking mask too. Notice that the result of 'longs' is one of (7*4) and (8*4).
        longs = (sa->sa_flags & SA_NOMASK)?7:8;
// To store the arguments of signal handler, we need to extend the space for them on the
// user state stack and varify the space (to allocate new page if not enough).
        *(&esp) -= longs;
        verify_area(esp,longs*4);
// In the user state stack, from the bottom to top, we store sa_restorer, signal number signr,
// signal blocking mask, register valuse eax, ecx, edx, eflags and the user program pointer.
        tmp_esp=esp;
        put_fs_long((long) sa->sa_restorer,tmp_esp++);
        put_fs_long(signr,tmp_esp++);
        if (!(sa->sa_flags & SA_NOMASK))
                put_fs_long(current->blocked,tmp_esp++);
        put_fs_long(eax,tmp_esp++);
        put_fs_long(ecx,tmp_esp++);
        put_fs_long(edx,tmp_esp++);
        put_fs_long(eflags,tmp_esp++);
        put_fs_long(old_eip,tmp_esp++);
        current->blocked |= sa->sa_mask;           // Add signal mask to blocked in PCB.
}

```

5.9.3 Information

5.9.3.1 Description of process signals

The signal in the process is a simple message used to communicate between processes. It is a simple integer value as shown in Table 5-3. For example, while a child process stop running or terminates, it will generate the signal SIGCHILD and send to the father to notify the current status of the child process.

To handling the signal received, a process has two kinds of method. One way is to do nothing and the kernel takes the default action for the signal as listed in the right side of the table. The other method is to write a function or handler for the signal and to handle the signal by the program itself.

Table 5-3 Process signals

Value	Name	Description	Default Action
1	SIGHUP	(Hangup) The kernel generates this signal when the process has no controlling terminal or when the modem is hangup.Since daemons doesn't have controlling terminal, they often use this signal to require rereading configuration file.	(Abort) Hang up the controlling terminal or process.
2	SIGINT	(Interrupt) Interrupt from keyboard. Usually the terminal driver combines this signal with key stroke '^C'.	(Abort) Terminate the process.
3	SIGQUIT	(Quit) Interrupt from keyboard. Usually the terminal driver combines this signal with key stroke ^\\.	(Dump) Terminate the process and dump core.
4	SIGILL	(Illegal Instruction) Program error or executes a illegal instruction.	(Dump) Terminate the process and dump core.
5	SIGTRAP	(Breakpoint/Trace Trap) Breakpoint for debugging.	(Dump) Terminate the process and dump core.
6	SIGABRT	(Abort) Abort the execution, abnormal termination.	(Dump) Terminate the process and dump core.
6	SIGIOT	(IO Trap) Hardware error, equivalent to SIGABRT.	(Dump) Terminate the process and dump core.
7	SIGUNUSED	(Unused) Not used, bad system call.	(Abort) Terminate the process.
8	SIGFPE	(Floating Point Exception) FPU error.	(Dump) Terminate the process and dump core.
9	SIGKILL	(Kill) Terminate the program. This signal can not be captured or ignored. If you want to terminate a program immediately, send this signal.	(Abort) Terminate the process.
10	SIGUSR1	(User defined Signal 1) Available to processes.	(Abort) Terminate the process.
11	SIGSEGV	(Segmentation Violation) Invalid memory reference.	(Dump) Terminate the process and dump core.
12	SIGUSR2	(User defined Signal 2) Available to processes for IPC or other purposes.	(Abort) Terminate the process.
13	SIGPIPE	(Pipe) Write to pipe with no readers.	(Abort) Terminate the process.
14	SIGALRM	(Alarm) Generated when the alarm timer expired.	(Abort) Terminate the process.

15	SIGTERM	(Terminate) Require to terminate a process friendly. It is the default signal for kill(). This signal can be captured.	(Abort) Terminate the process.
16	SIGSTKFLT	(Stack fault on coprocessor)	(Abort) Terminate the process.
17	SIGCHLD	(Child) Send to father process and tell him the child process has stopped or terminated.	(Ignore) Ignore this signal.
18	SIGCONT	(Continue) Resume the stopped process to run state.	(Continue) Resume execution if stopped.
19	SIGSTOP	(Stop) Stop process execution. This signal can not be captured or ignored.	(Stop) Stop the process.
20	SIGTSTP	(Terminal Stop) Stop the process issued from terminal. This signal can be captured or ignored.	(Stop) Stop the process.
21	SIGTTIN	(TTY Input on Background) The background process try to do input operation from a terminal. This process will be stopped until receiving a SIGCONT signal.	(Stop) Stop the process.
22	SIGTTOU	(TTY Output on Background) The background process try to do output operation to a terminal. This process will be stopped until receiving a SIGCONT signal.	(Stop) Stop the process.

5.10 exit.c

5.10.1 Function

This program primarily describes the operation of the termination or exit of a process. The actions involve process' resources release, session (or process group) termination and system calls like killing a process, terminating a process and hang up a process. It also includes signal sending functions such as send_sig() and tell_father().

The release() function deletes the process pointer in the task array, release the memory page relating to the process task structure and finally rescheduling the task.

The kill_session() function kills all the process in the session by sending SIGHUP signal.

The kill() system call is used to send any valid signal to a process. Based on the argument 'pid', this system call can send signal to deferent process or process group. The comments in the program listed bellow has presented all the cases.

The program exit handling function do_exit() is invoked in the interrupt procedure of the exit system call. Firstly, it frees the memory pages occupied by the code and data segments of the current process. If the current process owns the child processes, the father of them will be set to init process (process 1). If one of the child process has already be in zombie state, a SIGCHLD signal will be send to init process for this child process. Secondly, it closes all the files, releases the terminal device and coprocessor device. If the current process is the header of the process group, then all process in the group will be terminated. Thirdly, the current process is set to zombie state, the exit code is set and a signal SIGCHLD is sent to its father process. Lastly, it does task rescheduling.

The system call waitpid() is used to hang up the current process until the child process specified by the pid value is terminated or a terminating signal is received, or a signal handler is need to be invoked. If the child process specified by pid has already exit, waitpid() returns immediately. All resource used by the child process will be released.

5.10.2 Comments

Program 5-8 linux/kernel/exit.c

```
/*
 *  linux/kernel/exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>
#include <signal.h>
#include <sys/wait.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/tty.h>
#include <asm/segment.h>

int sys_pause(void);      // System call causing the process to sleep. (L144 in sched.c)
int sys_close(int fd);    // System call closing the file. (L192 in fs/open.c)

//// Free the task slot and release the memory page occupied by the task p.
// Argument p is a task structure pointer. This function scan the task table task[] to
// find the specified task p. If found, it first free the task slot of p, then release the
// memory page occupied by the task structure and its kernel stack space, and lastly doing
// reschedule. If the task is not found, then the kernel goes panic.
// This function is invoked in the function sys_kill() and sys_waitpid().
void release(struct task_struct * p)
{
        int i;

        if (!p)
                return;
        for (i=1 ; i<NR_TASKS ; i++)          // Scan task table for the task p.
                if (task[i]==p) {
                        task[i]=NULL;         // Free the task slot.
                        free_page((long)p);   // Free the memory page.
                        schedule();           // Rescheduling (Really needed?).
                        return;
                }
        panic("trying to release non-existent task");
}

//// Send signal 'sig' to the specified task 'p'.
// Arguments 'sig' is the signal number; 'p' is the specified task structure pointer;
// 'priv' is a signal sending privilege, when set, a signal can be sent without care
// about the user property or level of the sending process. This function checks the
// correctness of the arguments and send the signal to the specified task if the constraint
// condition ('priv') is satisfied.
static inline int send_sig(long sig,struct task_struct * p,int priv)
{
        if (!p || sig<1 || sig>32)
                return -EINVAL;
// If the signal sending privilege is set, or the euid (effective uid) of the current
// process is the euid of the specified process, or the current process is the supervisor,
// then the signal 'sig' is sent to the process 'p'. That is the corresponding bit in the
// task's field 'signal' is set. The suser() is defined as (current->euid==0), and used to check
// if the process is owned by the root user.
        if (priv || (current->euid==p->euid) || suser())
                p->signal |= (1<<(sig-1));
        else
                return -EPERM;
        return 0;
}

//// Terminate the current session.
static void kill_session(void)
{
// The task pointer *p is first set to point to the last entry of the task table.
        struct task_struct **p = NR_TASKS + task;

// Scan the task pointer table. For the all tasks, except task 0, if the task's session
// number equals to that of the current task, then sends signal SIGHUP to the task.
        while (--p > &FIRST_TASK) {
                if (*p && (*p)->session == current->session)
                        (*p)->signal |= 1<<(SIGHUP-1);
        }
}

/*
 * XXX need to check permissions needed to send signals to process
 * groups, etc. etc.  kill() permissions semantics are tricky!
 */
//// The system call kill() can send any kind of signal to any process or process group,
// not only can just kill a process.
// Argument 'pid' is the specified processs's pid; 'sig' is the specified signal number.
// If pid is positive, then signal sig is sent to the process with pid.
// If pid equals 0, then sig is sent to every process in the process group of the current
// process.
// If pid equals -1, then sig is sent to every process except for process 1 (init).
// If pid is less than -1, then sig is sent to every process in the process group -pid.
// If sig is 0, then no signal is sent, but error checking is still performed.
// The system call function scans the task table and sends the signal to the tasks that
// satisfied the conditions. If pid equals 0, it means that the current process is the
// leader of a process group. Thus, the signal needs to be sent to every process in the
// process group of the current process.
int sys_kill(int pid,int sig)
{
```

```
62          struct task_struct **p = NR_TASKS + task;
63          int err, retval = 0;
64
65          if (!pid) while (--p > &FIRST_TASK) {
66                  if (*p && (*p)->pgrp == current->pid)
67                          if (err=send_sig(sig,*p,1))        // Force to send the signal.
68                                  retval = err;
69          } else if (pid>0) while (--p > &FIRST_TASK) {
70                  if (*p && (*p)->pid == pid)
71                          if (err=send_sig(sig,*p,0))
72                                  retval = err;
73          } else if (pid == -1) while (--p > &FIRST_TASK)
74                  if (err = send_sig(sig,*p,0))
75                          retval = err;
76          else while (--p > &FIRST_TASK)
77                  if (*p && (*p)->pgrp == -pid)
78                          if (err = send_sig(sig,*p,0))
79                                  retval = err;
80          return retval;
81  }
82
    //// Notifying the father process. Send signal SIGCHLD to the process pid. The child
    // process will be terminated or stopped in the default case.
    // If the father process is not found, then the operated process need to be freeed by
    // itself. But for the requirement of POSIX.1, if the father process terminated previously,
    // its child processes should be adopted by the init process (process 1).
83  static void tell_father(int pid)
84  {
85          int i;
86
87          if (pid)
    // Scan the process table to find the specified process of pid. Send the signal SIGCHLD to
    // the process pid, the child process will be stopped or terminated.
88                  for (i=0;i<NR_TASKS;i++) {
89                          if (!task[i])
90                                  continue;
91                          if (task[i]->pid != pid)
92                                  continue;
93                          task[i]->signal |= (1<<(SIGCHLD-1));
94                          return;
95                  }
96  /* if we don't find any fathers, we just release ourselves */
97  /* This is not really OK. Must change it to make father 1 */
98          printk("BAD BAD - no father found\n\r");
99          release(current);                    // Release the process itself if no father found.
100 }
101
    //// Program exits handling function, called in the system call sys_exit() at L137.
    // Argument 'code' is a error code.
102 int do_exit(long code)
103 {
104         int i;
105
```

```
// Free the page table entries corresponding to the memory pages occupied by the code and
// data segment of the current process. 'free_page_tables()' is at L105 in mm/memory.c
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
// If the current process has got child processes, then changes the father process of the
// child processes to be process init (process 1). If a child is in the state of zombie,
// then sends a signal SIGCHLD to the init process.
        for (i=0 ; i<NR_TASKS ; i++)
                if (task[i] && task[i]->father == current->pid) {
                        task[i]->father = 1;
                        if (task[i]->state == TASK_ZOMBIE)
           /* assumption task[1] is always init */
                                (void) send_sig(SIGCHLD, task[1], 1);
                }
// Close all files opened.
        for (i=0 ; i<NR_OPEN ; i++)
                if (current->filp[i])
                        sys_close(i);
// Synchronizing the working directory, root directory and the I node of the executing
// program, then release them.
        iput(current->pwd);
        current->pwd=NULL;
        iput(current->root);
        current->root=NULL;
        iput(current->executable);
        current->executable=NULL;
// If the current process is the session leader and has a controlling terminal, then
// release it.
        if (current->leader && current->tty >= 0)
                tty_table[current->tty].pgrp = 0;
// If the current process once used coprocessor, then clear the used flag.
        if (last_task_used_math == current)
                last_task_used_math = NULL;
// If the current process is the session leader, then kills all the processes in the session.
        if (current->leader)
                kill_session();
// Change the state of the current process to be ZOMBIE, this means that the current process
// has released the resources, and saved the exit code to be read for the father process.
        current->state = TASK_ZOMBIE;
        current->exit_code = code;
// Send signal SIGCHLD to the father process.
        tell_father(current->father);
// Rescheduling, the father process can deal with the other matters concerned.
        schedule();
        return (-1);    /* just to suppress warnings */
}

//// System call exit(). Terminating the process.
int sys_exit(int error_code)
{
        return do_exit((error_code&0xff)<<8);
}

```

```
// System call waitpid(). It suspends the execution of the current process until a child
// process as specified by the pid argument has exited, or until a signal is delivered whos
// action is to terminated the current process or to call a signal handling function. If a
// child process as requested by pid has already exited by the time of the call, the function
// returns immediately. Any system resources used by the child process aare freed.
// If pid great than 0, then which means to wait for the child process whose process Id
// equals to the vaule of pid.
// If pid equals 0, which means to wait for any child process whose process group ID is equal
// to that of the calling process.
// If pid is less than -1, which means to wait for any child process whose process group ID
// is equal to the absolute value of pid.
// If pid is equal -1, which means to wait for any child process.
// The value of 'options' is an OR of zero or more of the following constants:
// WNOHANG, which means to return immediately if no child has exited.
// WUNTRACED, which means to also return for children which are stopped, and whose status
// has not been reported.
// If 'stat_addr' is not NULL, waitpid store status information in the location pointed
// by 'stat_addr'.
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
        int flag, code;
        struct task_struct ** p;

        verify_area(stat_addr,4);
repeat:
        flag=0;
// Scan the task table begins from the last entry, skip the empty slot, current process
// and the child processes not belong to the current process.
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
                if (!*p || *p == current)            // skip the empty and current slot.
                        continue;
                if ((*p)->father != current->pid)    // skip the non current's children.
                        continue;
// The selected process p from the scanning is definitively the child of current process.
// If the waited pid great than 0, but it is not equal to the p's pid, which means the p
// is one of the other child of the current process, then we skip it.
                if (pid>0) {
                        if ((*p)->pid != pid)
                                continue;
// Otherwise, if the specified waited pid is equal 0, which means we are waiting for any
// child whose process group ID is equal to the current process. In this case, if the
// scanning process's group ID is not equal to that of the current process, then we skip.
                } else if (!pid) {
                        if ((*p)->pgrp != current->pgrp)
                                continue;
// Otherwise, if the pid is less than -1, which means we are waiting for any child process
// whose process group ID is equal to the absolute value of pid. In this case, if the
// scanning process's group ID is not equal to the absolute value of the pid, then we skip.
                } else if (pid != -1) {
                        if ((*p)->pgrp != -pid)
                                continue;
                }
// If the above three judgements are not correct, it means the current process is waiting
// for any of its child process. This is the case with pid equals -1.
// At this moment, the selected process p is exactly the waited process. Now, we handle
// this child process based upon its status.
                switch ((*p)->state) {
// In the case the child is in STOPPED status, if the flag WUNTRACED is not set, which means
// we needn't return immediately, then we continue to scan the other processes. If the flag
// is set, then we save the status information 0x7f at the location of '*stat_addr' and
// return with the value of child's pid. Here the status value of 0x7f will make the macro
// WIFSTOPPED() to be true. Refer to L14 in include/sys/wait.h
                        case TASK_STOPPED:
                                if (!(options & WUNTRACED))
                                        continue;
                                put_fs_long(0x7f,stat_addr);
                                return (*p)->pid;
// In the case the child is in ZOMBIE status, we first add up its running time in user
// mode and kernel mode to the current process (the father process) respectively, then we
// obtain the child's pid and exit code and release the child. Finally we return the exit
// code and child's pid.
                        case TASK_ZOMBIE:
                                current->cutime += (*p)->utime;
                                current->cstime += (*p)->stime;
                                flag = (*p)->pid;               // Save child's pid.
                                code = (*p)->exit_code;         // Get child's exit code.
                                release(*p);                    // Release the child.
                                put_fs_long(code,stat_addr);    // Store the exit code.
                                return flag;                    // Return the child's pid.
// If the child's status is not STOPPED and ZOMBIE, then we temporary set the flag = 1, which
// means we have got one child matched condition but in the state of running or sleeping, and
// will be used bellow.
                        default:
                                flag=1;
                                continue;
                }
        }
// When finished scanning, if the flag is set, which means there is at least one child
// that matches the condition (not stopped or zombie). At this moment, if the option
// WNOHANG is set, which means to return immediately if no child has exited, then we
// return right away, otherwise we set the status of current process to be INTERRUPTIBLE
// and do reschedule.
        if (flag) {
                if (options & WNOHANG)        // Return immediately if options = WNOHANG
                        return 0;
                current->state=TASK_INTERRUPTIBLE;
                schedule();
// When re-executing this process, if there is no signal received (except SIGCHILD), then
// repeat the operation, otherwise return with error code.
                if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                        goto repeat;
                else
                        return -EINTR;
        }
// If we couldn't find a matched child, then return with error code.
        return -ECHILD;
```

```
}

```

5.11 fork.c

5.11.1 Function

fork system call is used to create new (child) process. In the Linux system, all processes are child process of init process (process 1) except process 0, which is created manually in the system initialization period. fork system call is implemented in the file system_call.s as a label name of sys_fork. The fork.c program contains assistant functions for the sys_fork.

The two main functions are find_empty_process() and copy_process(). In file system_call.s, fork system call first invokes find_empty_process() to find and get a empty slot in the process table, then invokes copy_process() to duplicate the task structure of the current (father) process. The other two important functions are memory area verification function verify_area() and memory copy function copy_mem().

The copy_process() function is used to create and duplicate the code, data and environment information of the current process for new child process. In the procedure of copying, the main job is to set up the parameters in the new process's task structure. Firstly, the function allocates one page in the main memory area for storing the new task structure, then it duplicates the information in the current process's task structure to be used as a template. Secondly, it modifies the contents of the new task structure, sets the current process to be the father of the new process, clears signal map and resets the accounting fields. Thirdly, it sets up the contents of the TSS based on that of the current process.

Since the new child needs to return 0, we should set tss.eax = 0. The new child's kernel stack pointer tss.esp0 should be set to the location at the top of the new allocated memory page, which is also occupied by the task structure begin from the bottom of the page. The new child's kernel stack segment register tss.ss0 is set to equal to the segment selector of the kernel data segment. The tss.ldt is set to the index value of the relating ldt entry in the GDT. If the current process has been using the coprocessor, then the whole context of the coprocessor should be saved to the tss.i387 structure in the new task structure.

After the above actions, the copy_mem() function is called to set up the base and size limits of the new task, and duplicates the pages tables of the current process. If there are files opened in the current process (the father process), these files are also be kept opening in the new process. So the coresponding file open counts need to be increased by 1 respectively. Hereafter, the TSS and LDT descriptor entries for the new process is to be installed in the GDT. The base addresses of their descriptors are pointed to the tss and ldt fields in the new task structure.

Finally, the status of the new process is set to be runnable, and the pid of the new process is returned to the current process.

Figure 5-11 demonstrates the adjusting range and start address of the memory for the verify_area() function. Since the write_verify() function needs to operate with unit in pages (4096 bytes per page), the orignal start address of the verifying memory space should be adjusted to be right at the pages boundary, and the verifying memory size is also adjusted accordingly.

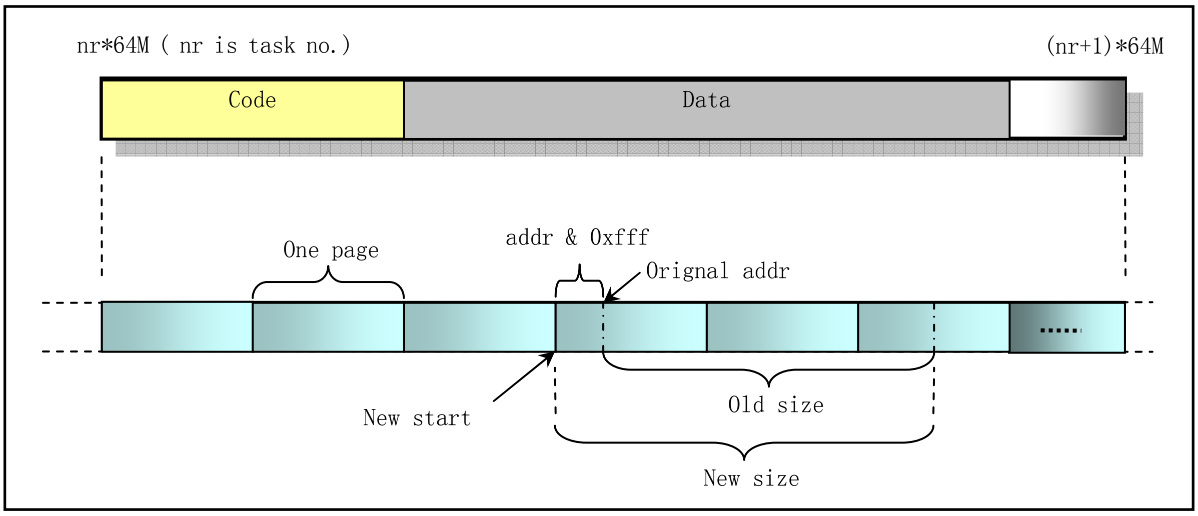

Figure 5-11 Adjusting the range and start address of the verifying memory.

5.11.2 Comments

Program 5-9 linux/kernel/fork.c

```
/*
 *  linux/kernel/fork.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'fork.c' contains the help-routines for the 'fork' system call
 * (see also system_call.s), and some misc functions ('verify_area').
 * Fork is rather simple, once you get the hang of it, but the memory
 * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
 */
#include <errno.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <asm/system.h>

extern void write_verify(unsigned long address);   // At L261 in mm/memory.c

long last_pid=0;            // The newest pid number, generated by get_empty_process().

// Function of verifying memory space before write operation.
// Checking the memory space ranging from 'addr' to 'addr + size' before write operation
// using page unit for the current process. Since the checking action is based on pages,
// it needs to find the start address 'start'of the page that contains the 'addr', and
// convert it to the CPU 4GB linear space by adding the base address of the process data
// segment. Afterwards, the write_verify() function is called in circle to do the real
// checking operation. If the page that will be written is readonly, then share verification
// and page copy operation are performend (Copy on Write).
void verify_area(void * addr,int size)
{
        unsigned long start;

        start = (unsigned long) addr;
// Adjust the beginning address 'addr' to the left boundary of the page that contains
// the 'addr', and resizing the verifying size accordingly.
// 'start & 0xfff' is used to obtains the offset in a page. By by adding the orignal size,
// the result 'size' is a adjusted size for the start address begins from a page boundary.
// The new start address is adjusted as 'start & 0xfffff000'.
        size += start & 0xfff;
        start &= 0xfffff000;
// Since the write_verify() function is operated in CPU 4GB linear space, the address 'start'
// needs first to be converted into it by adding the base address of the process.
        start += get_base(current->ldt[2]);
        while (size>0) {
                size -= 4096;
// Memory write verifying. If a page can not be written, then duplicates the page.
                write_verify(start);             // At L261 in mm/memory.c
                start += 4096;
        }
}

// Setup the base addresses and size limits of the code and data segments, copy page tables.
// Arguments nr is the new task number; p is the pointer of the new task structure.
int copy_mem(int nr,struct task_struct * p)
{
        unsigned long old_data_base,new_data_base,data_limit;
        unsigned long old_code_base,new_code_base,code_limit;

        code_limit=get_limit(0x0f);     // Get size limit in the code seg descriptor in LDT.
        data_limit=get_limit(0x17);     // Get size limit in the data seg descriptor in LDT.
        old_code_base = get_base(current->ldt[1]);    // Get code base of current process.
        old_data_base = get_base(current->ldt[2]);    // Get data base of current process.
        if (old_data_base != old_code_base)
                panic("We don't support separate I&D");   // Instruction & Data area.
        if (data_limit < code_limit)
                panic("Bad data_limit");
        new_data_base = new_code_base = nr * 0x4000000;   // New base = Task nr * 64MB.
        p->start_code = new_code_base;
        set_base(p->ldt[1],new_code_base);            // Setup base address for new task.
        set_base(p->ldt[2],new_data_base);
// Copy page table entries from the current process space to new task space. If an error
// occurred during copying, then release them all and return error code.
        if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
                free_page_tables(new_data_base,data_limit);
                return -ENOMEM;
        }
        return 0;
}

/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
// Copy and manipulate the new task structure from the current process.
// Argument 'nr' is the new task number, which is the return value of find_empty_process().
// 'none' is the return address pushed onto the stack when invoking sys_call_table in file
// system_call.s
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                long ebx,long ecx,long edx,
                long fs,long es,long ds,
                long eip,long cs,long eflags,long esp,long ss)
{
        struct task_struct *p;
        int i;
        struct file *f;

// Get one free page from main memory area for the new task structure and its kernel stack
// space. Put the new task structure pointer into task table in slot nr. Copy the contents
// of current process's task structure into the new task structure.
        p = (struct task_struct *) get_free_page();
        if (!p)
                return -EAGAIN;
        task[nr] = p;
        *p = *current;          /* NOTE! this doesn't copy the supervisor stack */
// We first set the status of the new task to be uninterruptible wait state. setup the
// new process pid to be 'last_pid', which is a global variable evaluated in the function
// find_empty_process().
        p->state = TASK_UNINTERRUPTIBLE;
        p->pid = last_pid;              // New process ID.
        p->father = current->pid;       // Set the fahter.
        p->counter = p->priority;
        p->signal = 0;                  // Reset the signal map. No signals now.
        p->alarm = 0;                   // Alarm timer (ticks).
        p->leader = 0;                  /* process leadership doesn't inherit */
        p->utime = p->stime = 0;        // The accounting values.
        p->cutime = p->cstime = 0;      // Child process accounting values.
        p->start_time = jiffies;        // Current system time.
// The following code initializes the TSS for the new task. Since the kernel allocate
// one memory page for the new task, thus, the tss.esp0 right points to the top of the
// page. ss0:esp0 is used as the kernel mode stack for the new task.
        p->tss.back_link = 0;
        p->tss.esp0 = PAGE_SIZE + (long) p;  // Kernel mode stack pointer.
        p->tss.ss0 = 0x10;              // Stack segment selector (same as data segment).
        p->tss.eip = eip;
        p->tss.eflags = eflags;
        p->tss.eax = 0;                 // This is the reason why child return 0.
        p->tss.ecx = ecx;
        p->tss.edx = edx;
        p->tss.ebx = ebx;
// The new task duplicates the stack contents of the father completely, thus, we require
// that the stack of the task 0 to be clean for the sake of this action.
        p->tss.esp = esp;
        p->tss.ebp = ebp;
        p->tss.esi = esi;
        p->tss.edi = edi;
        p->tss.es = es & 0xffff;     // Segment register has only 16 bits.
        p->tss.cs = cs & 0xffff;
        p->tss.ss = ss & 0xffff;
        p->tss.ds = ds & 0xffff;
        p->tss.fs = fs & 0xffff;
        p->tss.gs = gs & 0xffff;
// Install the selector of the LDT descriptor. The LDT descriptor is in the GDT.
        p->tss.ldt = _LDT(nr);
        p->tss.trace_bitmap = 0x80000000;
// If the current process has used coprocessor, then its context should be saved into
// tss.i387 structure. The instruction 'clts' is used to clear the task switched flag TS
// in the control register CR0. The TS flag helps to determine when the context of the
// coprocessor does not match that of the task being executed by the CPU. Whenever a task
// switched, the CPU sets the flag TS. If the TS set, then each 'ESC' instructions causes
// exception 7 and is captured. The 'WAIT' instruction also causes exception 7 if both TS
// and MP are set. Thus, when a task switch is happened after one instruction 'ESC', the
// context of the coprocessor may need to be saved before executing a new 'ESC' instruction.
// The exception handling procedure should save the context of the coprocessor and reset TS.
// 'fnsave' instruction is used to save the context of the coprocessor into the memory area
// (here is the tss.i387) specified by the target operand.
        if (last_task_used_math == current)
                __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
// Setup base addresses and limits in code and data segment descriptors for the new task.
// Duplicate the page tables from that of the current process. If an error occurred, then
// clear the task table slot for the new task, and free the page occupied by the new task
// structure.
        if (copy_mem(nr,p)) {          // Error occurred if the return value is non-zero.
                task[nr] = NULL;
                free_page((long) p);
                return -EAGAIN;
        }
// If there is files opened in the fahter process, then increasing the counts by 1 for each
// openning files respectively.
        for (i=0; i<NR_OPEN;i++)
                if (f=p->filp[i])
                        f->f_count++;
// Increasing current (father) process's counts of the pwd, root and executable by 1 each.
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
        if (current->executable)
                current->executable->i_count++;
// Install the TSS and LDT segment descriptors in GDT for the new task. The limits of the
// two segments are set to be 104 bytes equally. Refer to L52-66 in include/asm/system.h.
// Notice that the task register TR will be loaded by the CPU automatically.
```

```
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        p->state = TASK_RUNNING;                /* do this last, just in case */
        return last_pid;                        // Return the new task pid to the father.
}

// Generating a non existing pid and find an empty slot (task number) in the task table
// for the new task. the new pid is stored in a global variable 'last_pid'.
int find_empty_process(void)
{
        int i;

        repeat:
// If the value of 'last_pid' is out of the range of a positive integer after increased
// by 1, then recycling the pid number from 1 again. Afterwards, check the new last_pid
// to see if it has been used by any tasks in the system, otherwise repeat the opeartion
// above to get a new one.
                if ((++last_pid)<0) last_pid=1;
                for(i=0 ; i<NR_TASKS ; i++)
                        if (task[i] && task[i]->pid == last_pid) goto repeat;
// Scan the task table to get and return an empty slot index (task number).
        for(i=1 ; i<NR_TASKS ; i++)             // Slot 0 is excluded.
                if (!task[i])
                        return i;
// If all the 64 slots in the task table are all occupied, return with an error code.
        return -EAGAIN;
}

```

5.11.3 Information

5.11.3.1 Task State Segment (TSS)[1]

To provide efficient, protected multitasking, the 80386 employs several special data structures. It does not, however, use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures. The registers and data structures that support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later. In addition to the simple task switch, the 80386 offers two other task-management features:

1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the

[1] Copied and modified from Intel 80386 programming reference manual.

interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.

2. With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

All the information the processor needs in order to manage a task is stored in a special type of segment, a task state segment (TSS). Figure 5-12 shows the format of a TSS for executing 80386 tasks.

31 23	15 7 0	
I/O MAP BASE	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LDT selector	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
INSTRUCTION POINTER (EIP)		20
PAGE DIRECTORY REGISTER CR3 (PDBR)		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	08
ESP0		04
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	BACK LINK TO PREVIOUS TSS	00

Figure 5-12 Intel 80386 32-bit Task State Segment

The fields of a TSS belong to two classes:

1. A dynamic set that the processor updates with each switch from the task. This set includes the fields that store:
   - ♦The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).

- The segment registers (ES, CS, SS, DS, FS, GS).
- The flags register (EFLAGS).
- The instruction pointer (EIP).
- The selector of the TSS of the previously executing task (updated only when a return is expected).

2. A static set that the processor reads but does not change. This set includes the fields that store:
   - The selector of the task's LDT.
   - The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
   - Pointers to the stacks for privilege levels 0-2.
   - The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs . (Refer to Chapter 12 for more information on debugging.)
   - The I/O map base (refer to Chapter 8 for more information on the use of the I/O map).

Task state segments may reside anywhere in the linear space. The only case that requires caution is when the TSS spans a page boundary and the higher-addressed page is not present. In this case, the processor raises an exception if it encounters the not-present page while reading the TSS during a task switch. Such an exception can be avoided by either of two strategies:

1. By allocating the TSS so that it does not cross a page boundary.
2. By ensuring that both pages are either both present or both not-present at the time of a task switch. If both pages are not-present, then the page-fault handler must make both pages present before restarting the instruction that caused the task switch.

The state segment, like all other segments, is defined by a descriptor. The task register (TR) identifies the currently executing task by pointing to the TSS. The instructions LTR and STR are used to modify and read the visible portion of the task register.

The I/O instructions that directly refer to addresses in the processor's I/O space are IN, INS, OUT, OUTS. The 80386 has the ability to selectively trap references to specific I/O addresses. The structure that enables selective trapping is the I/O Permission Bit Map in the TSS segment. The I/O permission map is a bit vector. The size of the map and its location in the TSS segment are variable. The processor locates the I/O permission map by means of the I/O map base field in the fixed portion of the TSS. The I/O map base field is 16 bits wide and contains the offset of the beginning of the I/O permission map. The upper limit of the I/O permission map is the same as the limit of the TSS segment.

In protected mode, when it encounters an I/O instruction (IN, INS, OUT, or OUTS), the processor first checks whether CPL <= IOPL (in EFLAGS) . If this condition is true, the I/O operation may proceed. If not true, the processor checks the I/O permission map.

Each bit in the map corresponds to an I/O port byte address; for example, the bit for port 41 is found at I/O map base + 5, bit offset 1. The processor tests all the bits that correspond to the I/O addresses spanned by an I/O operation; for example, a doubleword operation tests four bits corresponding to four adjacent byte addresses. If any tested bit is set, the processor signals a general protection exception. If all the tested bits are zero, the I/O operation may proceed.

It is not necessary for the I/O permission map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had one bits in the map. For example, if TSS limit is equal to I/O map base + 31, the first 256 I/O ports are mapped; I/O operations on any port greater than 255 cause an exception. If I/O map base is greater than or equal to TSS limit, the TSS segment has no I/O permission map, and all I/O instructions in the 80386 program cause exceptions when CPL > IOPL. In the case of Linux 0.11 kernel, The I/O map base is set to be 0x8000, greater than that of the TSS limit (104 bytes). So the TSS has no I/O permission map.

Because the I/O permission map is in the TSS segment, different tasks can have different maps. Thus, the operating system can allocate ports to a task by changing the I/O permission map in the task's TSS.

In Linux 0.11, SS0:ESP0 in the TSS is used to store the kernel mode stack pointer of the task. Whenever a task enter into the kernel, the initial location of the kernel mode stack pointer is not changed, SS0:ESP0 will remain points to the top boundary of the memory page used by the task data structure. SS1:ESP1 and SS2:ESP2 are used as stack pointers for code running under level 1 and level 2. The Linux kernel does not use these two levels. The user mode stack pointer is stored in registers SS:ESP.

5.12 sys.c

5.12.1 Function

sys.c program includes implementations of many system calls. A few of the system calls have not been implementated in kernel 0.11 yet, and only renturn an error code -ENOSYS. A simple descriptions of all the system calls can refer to file include/linux/sys.h.

There are various kind of identifiers (IDs) in the program, including process ID (pid), process group ID (pgrp or pgid), user ID (uid), user group ID (gid), real user ID (ruid), real group ID (rgid), effective user ID (euid), effective group ID (egid) and session ID. Understanding them is important to read the code.

A user has a user ID (uid) and a group ID (gid), which are set in the passwd file for the user, and usually called real user ID (ruid) and real group ID (rgid). For each file in a file system, the I-node contains the uid and gid, which specify the owner of the file and what user group the file belongs to, and are used for the file access permission. In addition, a process contains three kinds of IDs in its task structure as shown in Table 5-4.

Table 5-4 User ID and group ID related to process.

Type	User ID	Group ID
Process's	uid - User ID, specifies the owner of the process.	gid - Group ID, specifies the user group of the process.
Effective	euid - Effective user ID, indicates the file access permission.	egid - Effective goup Id, indicates the file access permission.
Saved	suid - Saved user ID. When the set-user-ID flag of a file is set, the suid is equal to the uid of the file, otherwise the suid	sgid - Saved group ID. When the set-group-ID flag of a file is set, the sgid is equal to the gid of the file, otherwise the sgid equals the

equals the euid of the process.	egid of the process.

The uid and gid are that of the owner of the process, and they are actually the real user ID (ruid) and real group ID (rgid) of the process. User root (supervisor) may use functions set_uid() and set_gid() to modify them. The effective user ID (euid) and effective group ID (egid) are used to check the file access permission.

Saved user ID (suid) and saved group ID (sgid) are used to access file, which has the set-user-ID or set-group-ID flag set. When running a program, the process's euid is usually equal to the uid, and process's egid equals to gid. Thus, the process can only access the files which owns by the same euid and egid, or other files permitted. But, if a file's set-user-ID flag is set, then the euid of the process will be set to equal to the uid of the file owner. Therefore the process are permitted to access the restricted file and the uid of the file owner is saved in suid. In the same reason, the set-group-ID flag has the alike property and be operated in the similar method.

If, for example, the owner of a program file is user root (value is 0) and its set-user-ID flag is set, then, when the program is executed in a process, the process's effective user ID (euid) will be set to be root's ID (0). Thus, the process owns the root authority. A practical example is the line command passwd owned by user root. This command is a set-user-ID program and, therefore, allows normal users modifying their login passwords. Since the command needs to modify the password in the file /etc/passwd, which is also owned by the root, thus the passwd command executed by a normal user needs set-user-ID flag to be set.

In addition, A process has its own IDs, the process ID (pid), the process group ID (pgrp or pgid) and the session ID (session). These three IDs are used to indicate the relationship between processes and have nothing to do with user ID and user group ID.

5.12.2 Comments

Program 5-10 linux/kernel/sys.c

```
/*
 *  linux/kernel/sys.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>

#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <sys/times.h>
#include <sys/utsname.h>

// Return current date and time.
// Unimplemented system call and always return -1 and set 'errno' to ENOSYS.
int sys_ftime()
{
        return -ENOSYS;                         // Not implemented yet.
}

// Unimplemented system call. Obsoleted.
int sys_break()
{
        return -ENOSYS;
}

// The ptrace system call provides a means by which a parent process may observe and control
// the execution of another process, and examine and change its core image and registers.
// It is primarily used to implement breakpoint debugging and system call tracing.
int sys_ptrace()
{
        return -ENOSYS;
}

// Change and print terminal line settings.
int sys_stty()
{
        return -ENOSYS;
}

// Get terminal line settings. Obsoleted.
int sys_gtty()
{
        return -ENOSYS;
}

// Change the name or location of a file.
int sys_rename()
{
        return -ENOSYS;
}

// Unimplemented system call. Obsoleted.
int sys_prof()
{
        return -ENOSYS;
}

// Set real and/or effective group ID of the current process.
// Unprivileged users may only interchange the real group ID and the effective group ID,
// and may only set the effective group ID to the saved group ID.
int sys_setregid(int rgid, int egid)
{
        if (rgid>0) {
                if ((current->gid == rgid) ||
                    suser())
                        current->gid = rgid;
                else
                        return(-EPERM);
        }
        if (egid>0) {
                if ((current->gid == egid) ||
                    (current->egid == egid) ||
                    (current->sgid == egid) ||
                    suser())
                        current->egid = egid;
                else
                        return(-EPERM);
        }
        return 0;
}

// Set group identity (gid). Set the effective group ID of the current process. If the
// the caller is the superuser, the real and saved group ID's are also set.
int sys_setgid(int gid)
{
        return(sys_setregid(gid, gid));
}

// Switch process accounting of or off.
int sys_acct()
{
        return -ENOSYS;
}

// Map any physical memory area into process's virtual space.
int sys_phys()
{
        return -ENOSYS;
}

int sys_lock()
{
        return -ENOSYS;
}

int sys_mpx()
{
        return -ENOSYS;
}

int sys_ulimit()
{
        return -ENOSYS;
}

// Return the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.
// If tloc is non-NULL, the return value is also stored in the memory pointed to by it.
int sys_time(long * tloc)
{
        int i;

        i = CURRENT_TIME;
        if (tloc) {
                verify_area(tloc,4);            // Verify the memory space. (4 bytes)
                put_fs_long(i,(unsigned long *)tloc); // Stored in tloc in user space.
        }
        return i;
}

/*
 * Unprivileged users may change the real user id to the effective uid
 * or vice versa.
 */
// setreuid sets real and effective user IDs of the current process. Unprivileged users
// may only set the real user ID to the real user ID or the effective user ID, and may
// only set the effective user ID to the real user ID, the effective user ID or the saved
// user ID.
// If the real user ID is set or the effective user ID is set to a value not equal to the
// previous real user ID, the saved user ID will be set to the new effective user ID.
// Completely  analogously, setregid sets real and effective group ID's of the current
// process, and all of the above holds with "group" instead of "user".
int sys_setreuid(int ruid, int euid)
{
        int old_ruid = current->uid;

        if (ruid>0) {
                if ((current->euid==ruid) ||
                    (old_ruid == ruid) ||
                    suser())
                        current->uid = ruid;
                else
                        return(-EPERM);
        }
        if (euid>0) {
                if ((old_ruid == euid) ||
                    (current->euid == euid) ||
                    suser())
                        current->euid = euid;
                else {
                        current->uid = old_ruid;
                        return(-EPERM);
                }
        }
        return 0;
}

// setuid() sets the effective user ID of the current process. Unprivileged user may use
// setuid() to set the effective user ID to the saved user ID, or real user ID. If the
// caller is superuser, then the real, effective and saved user ID are all set to specified
// uid.
int sys_setuid(int uid)
{
        return(sys_setreuid(uid, uid));
}

```

```
    // stime sets the system's idea of the time and date. Time, pointed to by tptr, is
    // measured in seconds from 00:00:00 GMT January 1, 1970. stime() may only be executed by
    // the super user.
148 int sys_stime(long * tptr)
149 {
150         if (!suser())
151                 return -EPERM;
152         startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
153         return 0;
154 }
155
    // Get process times. Store the current process times user time (utime), sys time (stime),
    // child user time (cutime) and child sys time (cstime) in the struct tms that tbuf points to.
156 int sys_times(struct tms * tbuf)
157 {
158         if (tbuf) {
159                 verify_area(tbuf,sizeof *tbuf);
160                 put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);
161                 put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);
162                 put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime);
163                 put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime);
164         }
165         return jiffies;
166 }
167
    // brk sets the end of the data segment to the value specified by 'end_data_seg', when that
    // value is reasonable, the system does have enough memory and the process does not exceed
    // its max data size. The value must be greater than the end of code and less than the 16KB
    // from the end the stack. The return value is the new end value of the data seg.
    // This system call is usually packaged into a function in the library libc and the return
    // value is different.
168 int sys_brk(unsigned long end_data_seg)
169 {
    // If the argument 'end_data_seg' greater than the end of code and less than the start
    // point of the stack - 16KB, then set the end of data segment in the current process.
170         if (end_data_seg >= current->end_code &&
171             end_data_seg < current->start_stack - 16384)
172                 current->brk = end_data_seg;
173         return current->brk;
174 }
175
176 /*
177  * This needs some heave checking ...
178  * I just haven't get the stomach for it. I also don't fully
179  * understand sessions/pgrp etc. Let somebody who does explain it.
180  */
    // setpgid sets the process group ID of the process specified by pid to pgid. If pid is
    // zero, the process ID of the current process is used. If pgid is zero, the process ID
    // of the process specified by pid is used. If setpgid is used to move a process from one
    // process group to another, both process groups must be part of the same session. In this
    // case, the pgid specifies an existing process group to be joined and the session ID of
    // that group must match the session ID of the joining process (L193).
    // getpgid returns the process group ID of the process specified by pid. If pid is zero,
    // the process ID of the current process is used.
    int sys_setpgid(int pid, int pgid)
    {
            int i;

            if (!pid)                         // If pid=0, uses pid of the current process.
                    pid = current->pid;
            if (!pgid)                        // If pgid=0, uses the current's pid as pgid.
                    pgid = current->pid;      // [Here is iffer from the description of POSIX]
            for (i=0 ; i<NR_TASKS ; i++)      // Scan task Table, search the process pid.
                    if (task[i] && task[i]->pid==pid) {
                            if (task[i]->leader)        // Already leader?
                                    return -EPERM;
                            if (task[i]->session != current->session)
                                    return -EPERM;
                            task[i]->pgrp = pgid;       // Set task's pgrp.
                            return 0;
                    }
            return -ESRCH;
    }

    // Return the process group ID of the current process. Same as getpgid(0).
    int sys_getpgrp(void)
    {
            return current->pgrp;
    }

    // setsid() creates a new session if the calling process is not a process group leader.
    // The calling process is the leader of the new session, the process group leader of the
    // new process group, and has no controlling tty. The process group ID and session ID of
    // the calling process are set to the PID of the calling process. The calling process will
    // be the only process in this new process group and in this new session.
    // setsid -- SET Session ID。
    int sys_setsid(void)
    {
    // If the current process is not a process group leader and is not owned by a superuser,
    // then return error code.
            if (current->leader && !suser())
                    return -EPERM;
    // Set the current process to be the session leader, and its session = pgrp = pid.
            current->leader = 1;
            current->session = current->pgrp = current->pid;
            current->tty = -1;                // The process has no controlling terminal.
            return current->pgrp;             // Return session ID.
    }

    // uname() returns system information in the structure pointed to by name.
    // The utsname struct is defined in sys/utsname.h
    int sys_uname(struct utsname * name)
    {
    // Here gives the information in the structure. This coding style will surely be changed.
            static struct utsname thisname = {
                    "linux .0","nodename","release ","version ","machine "
            };
            int i;

            if (!name) return -ERROR;         // Error if the buffer pointer is NULL.
            verify_area(name,sizeof *name);   // Verify the memory area for saving info.
            for(i=0;i<sizeof *name;i++)       // Copy the info to user data space.
                    put_fs_byte(((char *) &thisname)[i],i+(char *) name);
            return 0;
    }

    // Set the file creation mask to be mask & 0777, and return the orignal mask.
    int sys_umask(int mask)
    {
            int old = current->umask;

            current->umask = mask & 0777;
            return (old);
    }

```

5.13 vsprintf.c

5.13.1 Function

function vsprintf() is used to produce formatted output to the character string, and is one function in the printf() family. Since it is a function in the standard library now and with almost no relationship to the remain kernel code, you can refer to the library manual to understand the usage of it and might skip it now.

5.13.2 Comments

Program 5-11 linux/kernel/vsprintf.c

```
1  /*
2   *  linux/kernel/vsprintf.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
8  /*
9   * Wirzenius wrote this portably, Torvalds fucked it up :-)
10  */
   // Lars Wirzenius is one of the best friends of Linus at Helsinki university, and stayed in
   // the same office once. When developping the Linux operation system in 1991, Linus is not
   // very skillful with C language and know little about the variable-length argument facilities.
   // Thus, Lars Wirzenius helped programmed this code. This program is much robust and only
   // one bug founded 3 year later (Line 130 bellow). His homepage: http://liw.iki.fi/liw/
11
12 #include <stdarg.h>
13 #include <string.h>
```

```
14
15 /* we use this so that we can do without the ctype library */
16 #define is_digit(c)     ((c) >= '' && (c) <= '9')
17
   // Convert a string to value. The argument is a pointer to pointer to a string.
   // The returns is the converted value. In addition, the pointer steps forward.
18 static int skip_atoi(const char **s)
19 {
20         int i=0;
21
22         while (is_digit(**s))
23                 i = i*10 + *((*s)++) - '';
24         return i;
25 }
26
   // Defines the symbols for the converting type.
27 #define ZEROPAD 1               /* pad with zero */
28 #define SIGN    2               /* unsigned/signed long */
29 #define PLUS    4               /* show plus */
30 #define SPACE   8               /* space if plus */
31 #define LEFT    16              /* left justified */
32 #define SPECIAL 32              /* 0x */
33 #define SMALL   64              /* use 'abcdef' instead of 'ABCDEF' */
34
   // Divide operation.
   // Input: 'n' is a dividend, 'base' is a divisor.
   // The result 'n' is quotient. The return value is the remainder.
35 #define do_div(n,base) ({ \
36 int __res; \
37 __asm__("divl %4":"=a" (n),"=d" (__res):"" (n),"1" (0),"r" (base)); \
38 __res; })
39
   // Convert a integer to a string with specified base system.
   // Arguments: num - Integer value; base - system base; size - length of string;
                 precision - digital length; type - type option (Refer to L27-33).
   // Output: A pointer to the string str.
40 static char * number(char * str, int num, int base, int size, int precision
41         ,int type)
42 {
43         char c,sign,tmp[36];
44         const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
45         int i;
46
   // If the type indicates lowercase, then defines the lowercase character set.
   // If the type indicates left justified, then clear the flag 'ZEROPAD' in type.
   // If the carry base is out of the range 2 and 36 then exit.
47         if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
48         if (type&LEFT) type &= ~ZEROPAD;
49         if (base<2 || base>36)
50                 return 0;
   // If the type indicates the need of zero padded, the set variable c='0' (that is ''),
   // otherwise set c equals space character (c=' ').
   // If the type indicates it's a signed value and less than 0, then set sign character and
   // get the absolute value of the input integer. Otherwise if the type indicates a sign of
   // plus, then set a plus character to sign, and set sign to space character if indicated
   // or set to '0'.
           c = (type & ZEROPAD) ? '' : ' ' ;
           if (type&SIGN && num<0) {
                   sign='-';
                   num = -num;
           } else
                   sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
   // If it's a signed integer, then decrease its szie with 1. If it's a special convertion,
   // then minus 2 once more for hex (used for the prefix '0x') or minus 1 once more for the
   // octals ( for the prefix '0').
           if (sign) size--;
           if (type&SPECIAL)
                   if (base==16) size -= 2;
                   else if (base==8) size--;
   // If the integer is 0, then set temporary string array temp to '0', otherwise do the
   // convertion operation using specified base system.
           i=0;
           if (num==0)
                   tmp[i++]='';
           else while (num!=0)
                   tmp[i++]=digits[do_div(num,base)];
   // If the number of the digital charactors great than the value of precision, then extends
   // the length of precision to match the number digital charactors.
           if (i>precision) precision=i;
           size -= precision;

   // Here begins the real convertion and store the result into string 'str' temporarily.
   // If the 'type' does not contain flag 'ZEROPAD' and 'LEFT', then we first fill the space
   // charactor into 'str' with number of remain size. If it has sign symbol, then fill the
   // sign character.
           if (!(type&(ZEROPAD+LEFT)))
                   while(size-->0)
                           *str++ = ' ';
           if (sign)
                   *str++ = sign;
   // If the type indicates that it is a special conversion, the put a '0' at the beginning
   // of str for the ocatals or put '0x' for the hex.
           if (type&SPECIAL)
                   if (base==8)
                           *str++ = '';
                   else if (base==16) {
                           *str++ = '';
                           *str++ = digits[33];   // 'X'或'x'
                   }
   // If there is no 'LEFT' flag in the 'type', then fill with paddle charactors ('0' or
   // space character).
           if (!(type&LEFT))
                   while(size-->0)
                           *str++ = c;
   // At this moment, the variable 'i' contains the number of digitals for 'num'. If the
   // number is less than the size of precision, then fills 'str' with precison-i number
   // of character '0'.
           while(i<precision--)
                   *str++ = '';
   // Fill the 'str' with converted digitals, i totally.
           while(i-->0)
                   *str++ = tmp[i];
   // If the size is still great than 0, means there is a 'LEFT' flag in the type. So we
   // put space characters in the remain size of the 'str'.
           while(size-->0)
                   *str++ = ' ';
           return str;
   }

   // Output the formatted string.
   // The arguments fmt is the formatting string, args is variable arguments, buf is the
   // buffer for formatted string.
   int vsprintf(char *buf, const char *fmt, va_list args)
   {
           int len;
           int i;
           char * str;             // Used for temporarily store the string during conversion.
           char *s;
           int *ip;

           int flags;              /* flags to number() */

           int field_width;        /* width of output field */
           int precision;          /* min. # of digits for integers; max
                                      number of chars for from string */
           int qualifier;          /* 'h', 'l', or 'L' for integer fields */

   // First we points to buffer 'buf', then scan the formating string 'fmt', obtainning the
   // various converting flags.
           for (str=buf ; *fmt ; ++fmt) {
                   if (*fmt != '%') {
                           *str++ = *fmt;
                           continue;
                   }

                   /* process flags */
                   flags = 0;
                   repeat:
                           ++fmt;          /* this also skips first '%' */
                           switch (*fmt) {
                                   case '-': flags |= LEFT; goto repeat;
                                   case '+': flags |= PLUS; goto repeat;
                                   case ' ': flags |= SPACE; goto repeat;
                                   case '#': flags |= SPECIAL; goto repeat;
                                   case '': flags |= ZEROPAD; goto repeat;
                                   }

   // Get width field of the current argument, and put into the variable 'field_width'.
   // If the width field contains digitals, then set it to be the width value. If it is
   // character '*', it means the next argument specifies the witdh. Thus, we invoke the macro
   // va_arg to get the width value. If, at this moment, the width value is less than 0, then
   // it means there is a flag field (Left justified). Thereby we should append this flag to
   // the flag variable 'flag' and set field width with its absolute value.
                   /* get field width */
                   field_width = -1;
                   if (is_digit(*fmt))
                           field_width = skip_atoi(&fmt);
                   else if (*fmt == '*') {
                           /* it's the next argument */    // A bug. Should insert '++fmt;'
                           field_width = va_arg(args, int);
                           if (field_width < 0) {
                                   field_width = -field_width;
                                   flags |= LEFT;
                           }
                   }

   // Get the precision field. The precision field begins with '.'. The handling way is
   // similar to that of the width field. If the precision field contains digitals, then
   // set it to be the precision value. If it is a character '*', means the next argument
   // specifies the precision. So we invoke the macro va_arg to get the precision. If the
   // value is less than 0, we get its absolute as its precision value.
                   /* get the precision */
                   precision = -1;
                   if (*fmt == '.') {
                           ++fmt;
                           if (is_digit(*fmt))
                                   precision = skip_atoi(&fmt);
                           else if (*fmt == '*') {
                                   /* it's the next argument */
                                   precision = va_arg(args, int);
                           }
                           if (precision < 0)
                                   precision = 0;
                   }

   // The following code analyzes the conversion gualifier.
   // 'h', 'l', 'L' are qualifers for input length modifier.
                   /* get the conversion qualifier */
                   qualifier = -1;
                   if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
                           qualifier = *fmt;
                           ++fmt;
                   }

   // The following code analyzes the conversion specifier.
                   switch (*fmt) {
   // If the specifier is 'c', it means the corresponding argument is a character. At this
   // moment, if the flag indicates with no left justified, then we should put number of
   // 'width value -1' space characters ahead of the field. If the width value is still great
   // than 0, means left justified, then we put number of 'width value - 1' space characters
   // after the field.
                   case 'c':
                           if (!(flags & LEFT))
                                   while (--field_width > 0)
                                           *str++ = ' ';
                           *str++ = (unsigned char) va_arg(args, int);
                           while (--field_width > 0)
                                   *str++ = ' ';
                           break;

```

// If the specifier is character 's', it means the corresponding argument is a string. First,
// we obtain the length of the string. If it great than the precision field value, then we
// extend the precision value to be equal to the string length. If, at thim moment, the flag
// does not indicate left justified, then we should put number of 'field width - string len'
// of space characters ahead of the field. If the width value is still great than 0, means it
// needs left justified, then we put number of 'field width - string len ' space characters.

```
                case 's':
                        s = va_arg(args, char *);
                        len = strlen(s);
                        if (precision < 0)
                                precision = len;
                        else if (len > precision)
                                len = precision;

                        if (!(flags & LEFT))
                                while (len < field_width--)
                                        *str++ = ' ';
                        for (i = 0; i < len; ++i)
                                *str++ = *s++;
                        while (len < field_width--)
                                *str++ = ' ';
                        break;

```

// If the specifier is 'o', it means the corresponding argument needs convert to string
// in octals.

```
                case 'o':
                        str = number(str, va_arg(args, unsigned long), 8,
                                field_width, precision, flags);
                        break;

```

// If the specifier is 'p', it means the corresponding argument is a pointer. In this case,
// if the argument has no width field, the default width is 8, and need to append 0.

```
                case 'p':
                        if (field_width == -1) {
                                field_width = 8;
                                flags |= ZEROPAD;
                        }
                        str = number(str,
                                (unsigned long) va_arg(args, void *), 16,
                                field_width, precision, flags);
                        break;

```

// If the specifier is 'x' or 'X', it means the argument needs print in hex. 'x' indicates
// print in lowercase.

```
                case 'x':
```

```
            flags |= SMALL;
        case 'X':
            str = number(str, va_arg(args, unsigned long), 16,
                field_width, precision, flags);
            break;

// If the specifier is 'd', 'i' or 'u', they mean the argument is a integer. As 'd' and 'i'
// represents signed integer, a sign character need to appended.
        case 'd':
        case 'i':
            flags |= SIGN;
        case 'u':
            str = number(str, va_arg(args, unsigned long), 10,
                field_width, precision, flags);
            break;

// If the specifer is 'n', it means we need to store the number of converted characters into
// the location pointed by the corresponding argument. Firstly, we use va_arg() to get the
// argument pointer, then save the number of converted characters into the location.
        case 'n':
            ip = va_arg(args, int *);
            *ip = (str - buf);
            break;

// If the specifier is not character '%', it means there is an error in the formating string,
// then we put a '%' into the output string instead. If the specifier contains other more
// characters, they are put into the output string too.
        default:
            if (*fmt != '%')
                *str++ = '%';
            if (*fmt)
                *str++ = *fmt;
            else
                --fmt;
            break;
        }
    }
    *str = '\0';        // Append a NULL at the end of the converted string.
    return str-buf;     // Return the length of the converted string.
}

```

5.14 printk.c

5.14.1 Function

printk() used in kernel code and has the same operation as printf() implemented in the standard library. The reason to re-program it is that the kernel code is forbidden to modify the segment register fs, which is dedicated to use in user mode. printk() uses function svprintf()

to format the arguments and then, after saved the register fs, invokes tty_write() to display messages on the console screen.

5.14.2 Commnets

Program 5-12 linux/kernel/printk.c

```
/*
 *  linux/kernel/printk.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * When in kernel-mode, we cannot use printf, as fs is liable to
 * point to 'interesting' things. Make a printf with fs-saving, and
 * all is well.
 */
#include <stdarg.h>
#include <stddef.h>

#include <linux/kernel.h>

static char buf[1024];

extern int vsprintf(char * buf, const char * fmt, va_list args);

// print function used in kernel code exclusively.
int printk(const char *fmt, ...)
{
        va_list args;                   // va_list is really a character pointer.
        int i;

// Begin arguments processing. Output the formatted string into buffer. variable 'i' contains
// the length of the formatted string.
        va_start(args, fmt);
        i=vsprintf(buf,fmt,args);
        va_end(args);
// Save the segment register fs, push three arguments for tty_write() and call it.
// Afterwards restore the register fs and return the length of the string.
        __asm__("push %%fs\n\t"         // Save fs
                "push %%ds\n\t"
                "pop %%fs\n\t"          // Set fs = ds
                "pushl %0\n\t"          // Arg1: Length of string.
                "pushl $_buf\n\t"       // Arg2: Address of the string buffer.
                "pushl $0\n\t"          // Arg3: Channel No. (0).
                "call _tty_write\n\t"   // (L290 in file kernel/chr_drv/tty_io.c)
                "addl $8,%%esp\n\t"     // Discard 2 pushed args (buf, channel).
                "popl %0\n\t"           // Pop the string length as return value.
                "pop %%fs"              // Restore fs.
                ::"r" (i):"ax","cx","dx");
        return i;                       // The length of the string.
}
```

5.15 panic.c

5.15.1 Function

Function panic() is used to show the kernel error message and let the kernel code goes into deadly circle whenever kernel code goes wrong. This is a simple but efficient way to handle the kernel code bugs, and follow the prinicple of the UN*X systems: The simple, the best.

In the book 'Hitch hikers Guide to the Glaxy', written by Douglas Adams, one of the famous sentence is: Don't Panic!

5.15.2 Comments

程序 5-1 linux/kernel/panic.c

```
/*
 *  linux/kernel/panic.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This function is used through-out the kernel (includeinh mm and fs)
 * to indicate a major problem.
 */
#include <linux/kernel.h>
#include <linux/sched.h>

void sys_sync(void);    /* it's really int */    /* (L44 in file fs/buffer.c) */

volatile void panic(const char * s)
{
        printk("Kernel panic: %s\n\r",s);
        if (current == task[0])
                printk("In swapper task - not syncing\n\r");
        else
                sys_sync();
        for(;;);
}

```

5.16 Summary

This chapter has disscused the 12 programs in the kernel subdirectory, which are the most important files in the kernel. The main contents include system calls, process schedule, signal processing, process forking and terminating.