Embedded System Design
Practice 7

TaeWook Kim & SeokHyun Hong
Hanyang University

SOFTWARE INTERRUPT

Preparation for the VPOS kernel porting

1. Implement Startup code
2. UART Settings
3. TIMER Settings
4. Implement Hardware Interrupt Handler
   (1) UART Interrupt
5. Implement Software Interrupt Entering/Leaving Routine
6. Kernel compile + load kernel image in RAM

Contents

1. Software Interrupt
2. Scheduler of VPOS
3. SWI Entering Routine & Leaving Routine
4. Homework

SOFTWARE INTERRUPT

Software Interrupt

- **Definition**
  - Aborting the execution of a program with a command included in the program, and transferring control to another program
  - It is not a hardware interrupting
  - Using SWI instruction to generate an interrupt

- **Purpose**
  - Used to enter privileged mode to invoke kernel functions
    - User Mode → Privileged Mode(Supervisor Mode)

Hardware Interrupt vs. Software Interrupt

- **Comparison**

	Hardware Interrupt (IRQ/FIQ)	Software Interrupt (SWI)
Purpose	General interrupt handling	OS protection mode
Vector Table Offset	0x18	0x08
Exception Priority	4	6
Interrupt disabled?	Disable	Do not disable
Link Register Revise	lr = lr - 4	Unnecessary

Hardware Interrupt vs. Software Interrupt

- **Link Register Adjust**
  - Hardware Interrupt
    - Executes the currently executing instruction and processes the interrupt
    - The next command to run is pc - 4
    - The value of the link register must be modified via lr = lr -4
  - Software Interrupt
    - Recognize the swi instruction in the decode stage of the pipeline and generate a software interrupt
    - The command that the PC points to is the command following swi
    - No need to modify link register value

SWI

- **SWI Instruction**
  - Commands that generate software interrupts
  - Allows operating system routines to be invoked in privileged mode by changing the process mode to Supervisor Mode
  - Notation : SWI {<cond>} SWI_number
    - SWI_number : Used to indicate special function calls or features

SCHEDULER OF VPOS

Scheduler in VPOS

- **vk_scheduler()**
  - Scheduler functions in VPOS
    - Static priority Ready queue structure of 32 steps
    - Round-Robin scheduling is used within the same priority
    - By calling vk_scheduler () at regular intervals with a timer interrupt, the thread executes only for a specified time slice
  - Kernel Function
    - Must be called in Privileged mode or System mode instead of User mode
    - To use the kernel function, you must switch from user mode to privileged mode.

      ➔ Using **Software Interrupt**

Scheduler Call Sequence (VPOS)

1. **Save 'CS' in the 'swi_number' variable of the current thread's struct variable**

2. **Use “swi 0x00” to cause a software interrupt**

3. **Jump to vk_swi_classifier() function in SWI entry routine**

4. **Calling the kernel function by checking the value of the 'swi_number' variable**
   - For CS, vk_scheduler () is called

5. **Start Scheduling**

vk_swi_scheduler() & vh_swi()

1. **Save 'CS' in the 'swi_number' variable of the current thread's struct variable**
   1. Execute VPOS_SHELL()

```
/* initialization for thread */
race_var = 0;
pthread_create(&p_thread, NULL, VPOS_SHELL, (void *)NULL);
pthread_create(&p_thread_0, NULL, race_ex_1, (void *)NULL);
pthread_create(&p_thread_1, NULL, race_ex_0, (void *)NULL);
pthread_create(&p_thread_2, NULL, race_ex_2, (void *)NULL);
```

vpos/kernel/kernel_start.c

   2. Execute vk_swi_scheduler

```
if(strcmp(argv[0], commands[cmd].command_string)==0)
{
    pthread_create(&p_thread, NULL, commands[cmd].func, (void *)&argv[1]);
    printk("\n");
    vk_swi_scheduler();
    cmd_check = 1;
    break;
}
cmd++;
```

vpos/shell/vpsh.c

vk_swi_scheduler() & vh_swi()

1. **Save 'CS' in the 'swi_number' variable of the current thread's struct variable**
   3. Save 'CS' in the 'swi_number' variable of the structure variable of the currently executing thread

   4. Execute vh_swi()

```
void vk_swi_scheduler(void)
{
    unsigned temp;

    vk_current_thread->swi_number = CS;
    temp = (unsigned)vk_current_thread;
    vh_swi(temp);
}
```

vpos/hal/cpu/hal_swi_handler.c

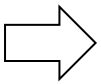
2. **Use “swi 0x00” to cause a software interrupt**

vk_swi_classifier()

3. Jump to vk_swi_classifier () function via SWI entry routine

SWI vector entry

```
vh_software_interrupt:
vh_entering_swi:
        bl      vk_swi_classifier
vh_leaving_swi:
```



```
void vk_swi_classifier(unsigned thread)
{
        unsigned number;
        vk_thread_t *vector;
        unsigned temp;
        int i;
        unsigned int *k=vk_save_swi_mode_stack_ptr;
        unsigned int *kk=vk_save_swi_current_tcb_bottom;
        printk("vk_swi_classifier switch up\n");
        vector=(vk_thread_t *)thread;
        number=vector->swi_number;
        switch(number)
        {
                case EI:
                        vector->interrupt_state = FALSE;
                        vh_enable_interrupt(vector);
                        break;
                case DI:
                        vector->interrupt_state = TRUE;
                        vh_disable_interrupt(vector);
                        break;
                case SC:
                        vh_save_thread_ctx((unsigned)vector->tcb_bottom);
                        break;
                case RC:
                        temp = (unsigned)vector->func;
                        vh_restore_thread_ctx((unsigned)vector->tcb_bottom);
                        break;
                case CS:
                        vk_scheduler();
                        break;
        }
}
```

SWI ENTERING ROUTINE & LEAVING ROUTINE

VPOS_kernel_main()

```
void VPOS_kernel_main( void )
{
    pthread_t p_thread, p_thread_0, p_thread_1, p_thread_2;

    /* static and global variable initialization */
    vk_scheduler_unlock();
    init_thread_id();
    init_thread_pointer();
    vh_user_mode = USER_MODE;
    vk_init_kdata_struct();

    vk_machine_init();
    set_interrupt();

    printk("%s\n%s\n%s\n", top_line, version, bottom_line);

    // Timer4 Test
    TIMER_test();

    /* initialization for thread */
    race_var = 0;
    pthread_create(&p_thread, NULL, VPOS_SHELL, (void *)NULL);
    pthread_create(&p_thread_0, NULL, race_ex_1, (void *)NULL);
    pthread_create(&p_thread_1, NULL, race_ex_0, (void *)NULL);
    pthread_create(&p_thread_2, NULL, race_ex_2, (void *)NULL);

    VPOS_start();

    /* cannot reach here */
    printk("OS ERROR: VPOS_kernel_main( void )\n");
    while(1){}
}
```

vpos/kernel/kernel_start.c

Uncomment

vk_serial_interrupt_handler()

```
void vh_serial_interrupt_handler(void)
{
    //printk("\nserial interrupt handler\n");
    vk_serial_push();
    vh_VIC1INTENCLEAR |= vh_VIC_UART1_bit;
    vh_VIC1INTENABLE |= vh_VIC_UART1_bit;
    vh_UINTP1 = 0xf;
}
```

Comment

vpos/hal/io.serial.c

vh_entering_swi

- **Implement SWI Entry Routines 1**
  1. Store the current sp value in a 'Vk_save_swi_mode_stack_ptr' variable (str)
  2. Stores pc, lr and general registers in the previous mode on the stack
     - Previous mode: Use ^
  3. Stores SPSR and lr on the stack
     - Save the SPSR to r0 with the mrs command and save r0 to the stack
  4. Disable IRQ Exception
     - Save the cpsr value in the r0 register and modify the interrupt mask bit
  5. Store the current sp value in the 'vk_save_swi_current_tcb_bottom' variable

vh_entering_swi

- **Implement SWI Entry Routines 2**

  6. Store the parameters in the r0 register for use by kernel functions
     - Use the ldr command. Add an offset to sp and load the r0 value stored in 2. on the stack
       - ldr r0, [sp, #offset]

  7. Jump to SWI handler
     - “bl vk_swi_classifier”

vh_entering_swi

```
vh_software_interrupt:
vh_entering_swi:
    str     sp, vk_save_swi_mode_stack_ptr
    stmfd   sp, {r14}^
    sub     sp, sp, #4
    stmfd   sp, {r13}^
    sub     sp, sp, #4
    stmfd   sp!, {r0-r12}

    mrs     r0, spsr_all
    stmfd   sp!, {r0, lr}
    mrs     r0, cpsr_all
    orr     r0, r0, #0x80
    msr     cpsr_all, r0

    str     sp, vk_save_swi_current_tcb_bottom
    ldr     r0, [sp, #8]
    bl      vk_swi_classifier
```

vpos/hal/cpu/HAL_arch_startup.S

vh_leaving_swi

- **Implement SWI return routine**
  1. Restore all registers that were saved on the stack
     - Must be stored in SPSR with msr command
  2. Return to original routine using link register (lr)

vh_leaving_swi

```
vh_leaving_swi:
    ldmfd   sp!, {r0, lr}
    msr     spsr_all, r0
    ldmfd   sp!, {r0-r12}
    ldmfd   sp, {r13}^
    add     sp, sp, #4
    ldmfd   sp, {r14}^
    add     sp, sp, #4
    movs    [redacted], [redacted]
```

vpos/hal/cpu/HAL_arch_startup.S

Compile & Upload

1. Compile (Ubuntu)

```
$ cd vpos
$ make clean; make
$ cp ./images/vpos.bin /tftpboot
```

2. Upload (Minicom)

```
$ tftp c0008000 vpos.bin
$ bootm c0008000
```

Result

```
vk_swi_classifier switch up
vk_swi_classifier switch up
vk_swi_classifier switch up
vk_swi_classifier switch up

Race condition value = 1200000

Shell>ls
vk_swi_classifier switch up

******************Command_List******************
- help
- ls
- debug
- thread
- temp
************************************************
vk_swi_classifier switch up

Shell>
```

Shell receives user input and creates a thread that handles user input

Shell calls scheduler to schedule the thread that we created by SWI

Thank you