운영체제의 기초 Project 1

전기정보공학부

2017-17088 박찬정

1. eOS 소스 코드 분석

(1) 시스템 초기화와 인터럽트 처리 루틴

\* 시스템 초기화 루틴

main 함수에서 인터럽트 벡터 테이블의 reset entry인 \_vector[0]로 jump하여 시스템 초기화를 완료할 때까지의 코드

우선 main 함수가 실행된다.

|  |
| --- |
| int main(int argc, char \*\*argv) {    PRINT("reset\n");    \_eflags = 0;    \_\_asm\_\_ \_\_volatile\_\_ ("\      mov $0x0, %%esp;\      jmp \*%0"      :: "r"(\_vector[0]));    /\* never return here \*/    return 0;  } |

\_eflags 변수를 0으로 둔다.

스택 포인터를 0으로 초기화하고, \_vector[0] 에 담긴 주소로 점프한다.

\_vector[0] 에는 무슨 값이 들어있을까? 이는 entry.S 파일을 확인하면 알 수 있다.

|  |
| --- |
| .data  .global \_vector  \_vector:    .long \_os\_reset\_handler    .long \_os\_reset\_handler    .long \_os\_reset\_handler    .long \_os\_irq\_handler |

위에서 \_vector[0]이 \_os\_reset\_handler를 갖고 있음을 알 수 있다.

\_os\_reset\_handler는 같은 파일 안에서 아래와 같이 정의되어 있다.

|  |
| --- |
| .text  .global \_os\_reset\_handler  \_os\_reset\_handler:    \_CLI    lea \_os\_init\_stack\_end, %esp    call \_os\_initialization    jmp \_os\_reset\_handler |

\_CLI의 내용은 emulator\_asm.h 파일내에서 확인할 수 있다.

|  |
| --- |
| /\* clear interrupt \*/  #define \_CLI \    movl $0, \_eflags; |

정리하자면, \_os\_reset\_handler의 동작은 아래와 같다.

우선 eflags에 0을 대입한다.

\_os\_init\_stack\_end 의 주소를 스택 포인터에 대입한다.

\_os\_initialization을 실행한다.

\_os\_initialization의 실행이 끝나면 os\_reset\_handler 본인의 처음 부분으로 점프한다.

\_os\_initialization은 core/main.c 안에 정의되어 있다.

|  |
| --- |
| void \_os\_initialization() {    \_os\_multitasking = 0;    eos\_disable\_interrupt();      // Initialize subsystems.    \_os\_init\_hal();    \_os\_init\_icb\_table();    \_os\_init\_scheduler();    \_os\_init\_task();    \_os\_init\_timer();    // Create idle task.    PRINT("creating idle task.\n");    eos\_create\_task(&idle\_task, (int32u\_t \*)idle\_stack, 8096, \_os\_idle\_task, NULL, LOWEST\_PRIORITY);    // After finishing initializations by kernel,    // give users a chance to do application specific initializations.    extern void eos\_user\_main();    eos\_user\_main();    // Start multitasking.    PRINT("finishing initialization. starting multitasking.\n");    \_os\_multitasking = 1;    eos\_enable\_interrupt();    eos\_schedule();    // After finishing all initializations, OS enters loop.    while(1) {}  } |

이 함수에서는 우선 \_os\_multitasking을 0으로 off하고, eos\_disable\_interrupt를 실행한다.

eos\_disable\_inturrupt 함수는 hal/linux/interrupt\_asm.S 파일에 정의되어 있다.

|  |
| --- |
| /\* disable irq and return previous status \*/  .global eos\_disable\_interrupt  eos\_disable\_interrupt:    mov \_eflags, %eax    \_CLI    ret |

이 함수는 현재의 \_eflags 값을 %eax 레지스터에 저장한 뒤 eflags에 값 0을 넣는 동작을 한다.

이제 \_os\_initialization에서 subsystem을 initialize 한다.

우선 \_os\_init\_hal 함수로 HAL을 초기화한다.

|  |
| --- |
| /\* intialize hardware dependent parts \*/  void \_os\_init\_hal() {    PRINT("initializing hal module.\n");    /\* initialize timer interrupt \*/    \_init\_timer\_interrupt();    /\* initiate interval timer by unmasking timer interrupt \*/    eos\_enable\_irq\_line(IRQ\_INTERVAL\_TIMER0);  } |

여기서는 timer inturrupt를 초기화하고, 해당 inturrupt의 irq 비트마스크를 on 한다.

둘째로, \_os\_init\_icb\_table 함수를 사용하여 interrupt control block table을 초기화한다.

함수와 관련 내용은 core/interrupt.c 파일에 정의되어 있다.

|  |
| --- |
| /\*   \* The ICB structure.   \* This represents a in-kernel status of an irq   \*/  typedef struct icb {    int8s\_t irqnum;   /\* the irq number of this ICB. \*/    void (\*handler)(int8s\_t irqnum, void \*arg); /\* the handler to handle this interrupt. \*/    void \*arg;    /\* the argument given to the handler when interrupt is occurred. \*/  } \_os\_icb\_t;  /\*   \* Table of ICB for all interrupts   \*/  \_os\_icb\_t \_os\_icb\_table[IRQ\_MAX];  void \_os\_init\_icb\_table() {    PRINT("initializing interrupt module.\n");    int8s\_t i;    for (i=0; i<IRQ\_MAX; i++) {      \_os\_icb\_t \*p = &\_os\_icb\_table[i];      p->irqnum = i;      p->handler = NULL;    }  } |

Icb는 irq number, interrupt 상태에서의 핸들러와 그 핸들러에 제공할 인자를 저장하고 있다.

그리고 이 icb의 array 형태로 icb table을 만들어 관리한다.

\_os\_init\_icb\_table에서는 \_os\_icb\_table를 순회하며 irq number를 부여하고, handler에 NULL 값을 넣어둔다.

셋째로, \_os\_init\_scheduler 함수를 사용하여 스케쥴러를 초기화한다.

관련 내용은 core/scheduler.c 파일에 정의되어 있다.

|  |
| --- |
| void eos\_trigger\_counter(eos\_counter\_t\* counter) {    PRINT("tick\n");  }  /\* Timer interrupt handler \*/  static void timer\_interrupt\_handler(int8s\_t irqnum, void \*arg) {    /\* trigger alarms \*/    eos\_trigger\_counter(&system\_timer);  }  void \_os\_init\_scheduler() {    PRINT("initializing scheduler module.\n");      /\* initialize ready\_group \*/    \_os\_ready\_group = 0;    /\* initialize ready\_table \*/    int8u\_t i;    for (i=0; i<READY\_TABLE\_SIZE; i++) {      \_os\_ready\_table[i] = 0;    }    /\* initialize scheduler lock \*/    \_os\_scheduler\_lock = UNLOCKED;  } |

여기서는 ready 상태의 프로세스 정보를 저장하는 \_os\_ready\_table을 초기화한다. 모두 0으로 두고, lock도 해제해둔다.

넷째로, \_os\_init\_task를 사용하여 현재 task의 상태를 초기화한다.

관련 내용은 core/task.c 파일에 저장되어 있다.

|  |
| --- |
| void \_os\_init\_task() {    PRINT("initializing task module.\n");    /\* init current\_task \*/    \_os\_current\_task = NULL;    /\* init multi-level ready\_queue \*/    int32u\_t i;    for (i = 0; i < LOWEST\_PRIORITY; i++) {      \_os\_ready\_queue[i] = NULL;    }  } |

현재 실행중인 task를 NULL로 두고, ready 상태의 프로세스를 관리하는 \_os\_ready\_queue의 내용을 모두 NULL로 둔다.

마지막으로 \_os\_init\_timer 함수를 사용하여 timer를 초기화시킨다.

|  |
| --- |
| void \_os\_init\_timer() {    eos\_init\_counter(&system\_timer, 0);    /\* register timer interrupt handler \*/    eos\_set\_interrupt\_handler(IRQ\_INTERVAL\_TIMER0, timer\_interrupt\_handler, NULL);  } |

eos\_init\_couter 함수를 사용하여 system\_timer를 0으로 초기화한다.

그리고 eos\_set\_interrupt\_handler를 사용하여 timer interrupt의 핸들러로 timer\_interrupt\_handler를 제공한다. 현재로서는 “tick” 이라는 문구를 출력하도록 되어 있다.

이후 eos\_create\_task 함수로 idle task 하나를 만든다(현 시점에서는 받은 내용을 print 해주는 함수로만 구현되어 있다).

그러면 kernel에서의 initialization이 종료된다.

이후로는 유저의 initialization을 수행하도록 하고, 멀티태스킹과 인터럽트를 활성화시킨 뒤 스케쥴러를 실행하고, 루프에 진입한다.

\* 인터럽트 처리 루틴

\_gen\_irq 함수에서 인터럽트 벡터 테이블의 irq entry인 \_vector[3]로 jump하여 인터럽트 처리를 완료할 때까지의 코드

hal/linux/emulator/intr.c 파일에서 시작하자.

|  |
| --- |
| void \_gen\_irq(int8u\_t irq) {    \_irq\_pending |= (0x1 << irq);    \_deliver\_irq();  } |

irq 인자를 받아서 \_irq\_pending 레지스터에 해당 번 째 비트를 1로 둔다.

이후 \_deliver\_irq 함수를 수행한다. 이 함수는 hal/linux/emulator/vector.c 파일에 있다.

|  |
| --- |
| /\* deliver an irq to CPU \*/  void \_deliver\_irq() {    //PRINT("\_eflags: %d, \_irq\_pending: 0x%x, \_irq\_mask: 0x%x\n", \_eflags, \_irq\_pending, \_irq\_mask);    if (\_irq\_pending & ~\_irq\_mask) {      if (\_eflags == 1) {        \_eflags = 0;        \_eflags\_saved = 1;        //PRINT("interrupted\n");        \_\_asm\_\_ \_\_volatile\_\_ ("\          push $resume\_eip;\          jmp \*%0;\        resume\_eip:"          :: "r"(\_vector[3]));      }    }  } |

우선 \_irq\_pending 레지스터에 1인 비트가 있어야 하고, 해당 비트가 \_irq\_mask에서 0이어야 한다. \_gen\_irq에서 \_irq\_pending 레지스터는 켜 두었으므로 해당 비트의 \_irq\_mask만 0이면 된다.

또한 \_eflags가 1임을 확인한다. 그러면 \_eflags를 끄고(인터럽트 루틴 중에는 인터럽트가 발생하면 안되므로) \_eflags\_saved에 1을 저장한다. 그리고 어셈블리로 적힌 코드를 수행하는데, 이는 \_vector[3] 에 있는 주소로 jump하도록 한다. \_vector[3]에 있는 주소는 interrupt handler의 주소이다. hal/linux/entry.S 에서 확인할 수 있다.

|  |
| --- |
| .data  .data  .global \_vector  \_vector:    .long \_os\_reset\_handler    .long \_os\_reset\_handler    .long \_os\_reset\_handler    .long \_os\_irq\_handler |

같은 파일 내에 \_os\_irq\_handler 가 정의되어 있다.

|  |
| --- |
| .global \_os\_irq\_handler  \_os\_irq\_handler:    pusha    push \_eflags\_saved    call \_os\_common\_interrupt\_handler    add $0x4,%esp    popa    \_IRET |

우선 pusha 명령어를 사용하여 모든 레지스터 값을 스택에 저장한다. Context change가 일어나는 과정이라고 할 수 있다. 여기에 eflag\_saved 값을 스택에 저장한다. 이는 이후 함수 수행을 위한 인자로 사용된다.

이후 \_os\_common\_interrupt\_handler를 불러서 인터럽트 루틴을 수행한다. 이는 core/interrupt.c 파일에 정의되어 있다.

|  |
| --- |
| void \_os\_common\_interrupt\_handler(int32u\_t flag) {    /\* get the irq number \*/    int32u\_t irq\_num = eos\_get\_irq();    if (irq\_num == -1) { return; }      /\* acknowledge the irq \*/    eos\_ack\_irq(irq\_num);      /\* restore the \_eflags \*/    eos\_restore\_interrupt(flag);    /\* dispatch the handler and call it \*/    \_os\_icb\_t \*p = &\_os\_icb\_table[irq\_num];    if (p->handler != NULL) {      //PRINT("entering irq handler 0x%x\n", (int32u\_t)(p->handler));      p->handler(irq\_num, p->arg);      //PRINT("exiting irq handler 0x%x\n", (int32u\_t)(p->handler));    }  } |

우선 eos\_get\_irq 함수로 인터럽트가 발생한 index를 확인한다. 만약 없다면(return 값이 -1)이라면 함수를 즉시 종료한다. 그렇지 않다면, eos\_ack\_irq로 해당 인터럽트를 받았음을 알리고(\_irq\_pending에서 해당 비트를 끈다), eos\_restore\_interrupt를 수행한다. 이 함수는 hal/linux/interrupt\_asm.S 파일에 정의되어 있다.

|  |
| --- |
| /\* restore irq status \*/  .global eos\_restore\_interrupt  eos\_restore\_interrupt:    mov 0x4(%esp), %eax    mov %eax, \_eflags    ret |

작동은 간단한데, 인자로 받은 값을 \_eflags에 집어넣는 것이다. 함수의 이름대로 이전의 \_eflags 상태로 저장해두었던 것을 복원하게 된다.

\_os\_commong\_interrupt\_handler로 돌아와서, 이제 icb를 관리하게 된다. 인터럽트가 발생한 icb를 가져와서 icb가 가진 handler를 icb가 가진 인자와 함께 실행시킨다. 물론 handler가 NULL이라면 아무것도 하지 않는다.

(2) 인터럽트 관리 모듈

\* eos\_ack\_irq()

|  |
| --- |
| /\* ack the specified irq \*/  void eos\_ack\_irq(int32u\_t irq) {    /\* clear the corresponding bit in \_irq\_pending register \*/    \_irq\_pending &= ~(0x1<<irq);  } |

irq 번호를 받아서, \_irq\_pending의 해당 번째 비트를 0으로 만든다.

Interrupt service routine에 진입한 뒤, 이제 handling을 시작하게 되면서 수행하는 함수이다.

\* eos\_get\_irq()

|  |
| --- |
| /\* get the irq number \*/  int32s\_t eos\_get\_irq() {    /\* get the highest bit position in the \_irq\_pending register \*/    int i;    for(i=31; i>=0; i--) {      if (\_irq\_pending & ~\_irq\_mask & (0x1<<i)) {        return i;      }    }    return -1;  } |

\_irq\_pending 에서 1인 비트가 있는지 확인하고, 있다면 해당 index를 반환한다. 없다면 -1을 반환한다.

\* eos\_disable\_irq\_line(int32u\_t irq)

|  |
| --- |
| /\* mask an irq \*/  void eos\_disable\_irq\_line(int32u\_t irq) {    /\* turn on the corresponding bit \*/    \_irq\_mask |= (0x1<<irq);  } |

\_irq\_mask의 특정 비트를 1로 만드는 함수이다. \_irq\_mask가 1이면 해당 index의 interrupt가 발생해도 handler를 수행하지 않는다.

\* eos\_enable\_irq\_line(int32u\_t irq)

|  |
| --- |
| /\* unmask an irq \*/  void eos\_enable\_irq\_line(int32u\_t irq) {    /\* turn off the corresponding bit \*/    \_irq\_mask &= ~(0x1<<irq);  } |

eos\_disable\_irq\_line 함수와 정확히 반대의 역할을 한다. \_irq\_mask의 특정 비트를 0으로 만든다. 이제 해당 index의 interrupt는 발생시에 handler를 수행한다.

\* eos\_disable\_interrupt

|  |
| --- |
| /\* disable irq and return previous status \*/  .global eos\_disable\_interrupt  eos\_disable\_interrupt:    mov \_eflags, %eax    \_CLI    ret |

현재 \_eflags 값을 %eax 레지스터에 저장하고, \_eflags에 0을 집어넣는다.

%eax에 넣은 이전 \_eflags 값은 자연스럽게 return 값이 된다.

\* eos\_enable\_interrupt

|  |
| --- |
| /\* enable irq by force \*/  .global eos\_enable\_interrupt  eos\_enable\_interrupt:    \_STI    ret |

\_eflags에 1을 집어넣는다.

\* eos\_restore\_interrupt

|  |
| --- |
| /\* restore irq status \*/  .global eos\_restore\_interrupt  eos\_restore\_interrupt:    mov 0x4(%esp), %eax    mov %eax, \_eflags    ret |

인자로 받은 값을 %eax 레지스터에 집어넣고, %eax 레지스터의 값을 \_eflags에 집어넣는다.

인자로 이전 \_eflags 값을 받으면 이름에 맞는 행동을 하게 된다.