Operating Systems

Interrupts

1

Interrupts and Interrupt Handling

- Interrupt and exceptions concepts
- Interrupt implementation details
- Interrupt handling in the OS

- Motivation: separate I/O activities from processing. I/O is generally slow, but it can proceed in parallel with hardware that supports interrupting the CPU to signal completion of some I/O activity.

2

Processing an Interrupt

- Jump to exception vector
- Disable further interrupts
  - Some counter-examples with interrupt priorities
- Save state to return to the code executing when the interrupt occurs
  - The processor will save only minimal values, leaving it to the OS to completely save state as necessary
- Jump to a specific memory location
  - Directly or indirectly arriving at the memory location with code to handle the specific interrupt
- Restore state and resume normal processing

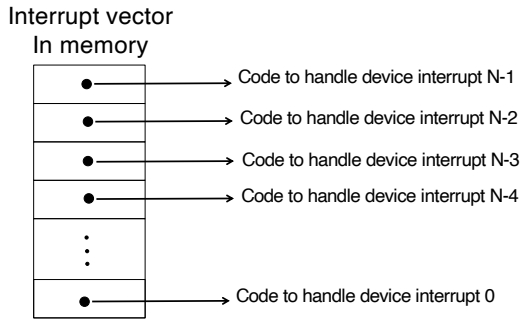
3

Vectored Interrupts

- Implementations vary, but in general an interrupt is identified by an Interrupt Request Number (IRQ)
  - Configured in different ways
- The IRQ is used as an index into a vector of function pointers
- interrupt_vector[IRQ] points to code to handle interrupt #IRQ

4

Vectored Interrupts



5

Exceptions

- Exceptions and Interrupts are realized in essentially the same way
- Exceptions such as page faults, illegal instructions transfer control to specified handler code
- More generally, the interrupt vector is an exception vector
  - Particularly on x86; ARM has two levels as we will see

6

Intel Exceptions and Interrupts

- The Intel Galileo has a unified vector of function pointers for both exceptions and IRQs
- Exception 0 indicates the “divide by zero” exception
- Hardware handles this just as if a device with IRQ 0 fired
  - Of course this means IRQ 0 can’t be used for a device

7

Interrupts in ARM

- The ARM processor has various states including User mode, System mode and exception modes
- Distinct exception states
  - Software Interrupt
  - Reset
  - Undefined Instruction
  - Abort (Pre-fetch or Data)
  - Interrupt (IRQ and FIQ)
- Everything is mapped to one of these exception states
- Top level handlers are one of these exception states

8

IRQ and FIQ

- ARM supports IRQs and Fast IRQs or FIQs
  - Xinu only supports IRQs
- Both IRQ and FIQ mode "bank" registers, meaning they are hidden until that mode is restored
  - r0 – r6 are visible across modes
  - IRQ mode banks the Stack Pointer (SP) and the Link Register (LR)
  - FIQ mode banks those as well as R8-R12
- FIQ mode can be used to optimize interrupt processing

9

Interrupts in ARM

- When a IRQ is raised, the ARM processor:
- Stops what it is doing, jumps to the IRQ handler location and switches to IRQ mode
- Disables further IRQs (not FIQs)
- Puts CPSR in SPSR
- Puts the current PC in LR
- Consults the exception vector and jumps to the appropriate handler
- For an IRQ, it jumps to the exception handler for IRQs, which is itself a table of handlers

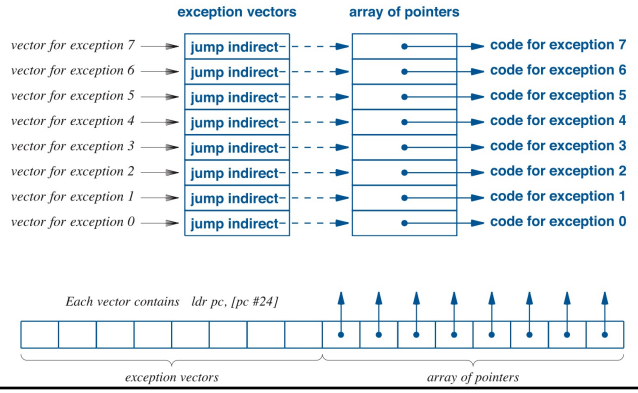
10

ARM Exception Vectors

- While Intel's exception vector contains pointers, the ARM exception vector contains instructions
- In practice, they are simply instructions that jump to an address but the implementation is different
  - ARM: PC = exception_vector[i]
  - Intel: PC = *exception_vector[i]
- Each exception vector entry is one 32-bit instruction, so they are structured as indirect branch instructions

11

Still an array of pointers in practice



12

intr.S

```
/*intr.S - enable, disable, restore, halt, pause, irq_except(ARM) */

#include <armv7a.h>

    .text
    .globl  disable
    .globl  restore
    .globl  enable
    .globl  pause
    .globl  halt
    .globl  irq_except
    .globl  irq_dispatch
    .globl  initevec
    .globl  expjmpinstr

/*-------------------------------------------------------------------------
 * disable  -  Disable interrupts and return the previous state
 *-------------------------------------------------------------------------
 */

disable:
    mrs     r0, cpsr        /* Copy the CPSR into r0                */
    cpsid   i               /* Disable interrupts                   */
    mov     pc, lr          /* Return the CPSR                      */
```

13

intr.S

```
/*-------------------------------------------------------------------------
 * restore  -  Restore interrupts to value given by mask argument
 *-------------------------------------------------------------------------
 */

restore:
    push    {r1, r2}        /* Save r1, r2 on stack                 */
    mrs     r1, cpsr        /* Copy CPSR into r1                    */
    ldr     r2, =0x01F00220
    and     r1, r1, r2      /* Extract flags and other important    */
    bic     r0, r0, r2      /*    bits from the mask                */
    orr     r1, r1, r0
    msr     cpsr_cfsx, r1   /* Restore the CPSR                     */
    pop     {r1, r2}        /* Restore r1, r2                       */
    mov     pc, lr          /* Return to caller                     */


/*-------------------------------------------------------------------------
 * enable  -  Enable interrupts
 *-------------------------------------------------------------------------
 */

enable:
    cpsie   i               /* Enable interrupts                    */
    mov     pc, lr          /* Return                               */
```

14

intr.S

```
/*-------------------------------------------------------------------------
 * initevec - Initialize the exception vector
 *------------------------------------------------------------------------- */
initevec:
        mrc     p15, 0, r0, c1, c0, 0 /* Read the c1-control register   */
        bic     r0, r0, #ARMV7A_C1CTL_V/* V bit = 0, normal exp. base   */
        mcr     p15, 0, r0, c1, c0, 0 /* Write the c1-control register  */
        ldr     r0, =ARMV7A_EV_START  /* Exception base address         */
        mcr     p15, 0, r0, c12, c0, 0/* Store excp. base addr. in c12  */
        ldr     r0, =ARMV7A_EV_START  /* Start address of exp. vector   */
        ldr     r1, =ARMV7A_EV_END    /* End address of exp. vector     */
        ldr     r2, =expjmpinstr      /* Copy the exp jump instr        */
        ldr     r2, [r2]              /*   into register r2             */
expvec:  str     r2, [r0]             /* Store the jump instruction     */
        add     r0, r0, #4            /*   in the exception vector      */
        cmp     r0, r1
        bne     expvec
        ldr     r0, =ARMV7A_EH_START  /* Install the default exception  */
        ldr     r1, =ARMV7A_EH_END    /*   handler for all exceptions   */
        ldr     r2, =defexp_handler
exphnd:  str     r2, [r0]
        add     r0, r0, #4
        cmp     r0, r1
        bne     exphnd
        ldr     r0, =ARMV7A_IRQH_ADDR /* Install the IRQ handler to      */
        ldr     r1, =irq_except       /*   override the default         */
        str     r1, [r0]              /*   exception handler            */
        mov     pc, lr
```

15

intr.S

```
/*-------------------------------------------------------------------------
 * irq_except  -  Dispatch an IRQ exception to higher level IRQ dispatcher
 *-------------------------------------------------------------------------
*/

irq_except:
    sub     lr, lr, #4      /* Correct the return address           */
    srsdb   sp!, #19        /* Save return state on the supervisor  */
                            /*   mode stack                         */
    cps     #19             /* Change to supervisor mode            */
    push    {r0-r12, lr}    /* Save all registers                   */
    bl      irq_dispatch    /* Call IRQ dispatch                    */
    pop     {r0-r12, lr}    /* Restore all registers                */
    rfeia   sp!             /* Return from the exception using info */
                            /*   stored on the stack                */

/*-------------------------------------------------------------------------
 * defexp_handler - Default Exception handler
 *-------------------------------------------------------------------------
 */

defexp_handler:
    ldr     r0, =expmsg1
    mov     r1, lr
    bl      kprintf
    ldr     r0, =expmsg2
    bl      panic
```

16

Assigning Device Numbers

- Manual configuration with jumpers and switches
  - This is where I came in. Error-prone even when you knew what you were doing…
- Automatic assignment at boot time
  - PCI brought the ability to assign IRQ numbers to devices after interrogating them
- Dynamic assignment for runtime pluggable devices
  - USB master driver loaded at initialization time
  - Gets an interrupt when a new device is connected
  - Interrogates device, loads appropriate driver
  - Vectors through another table for device interrupt handlers

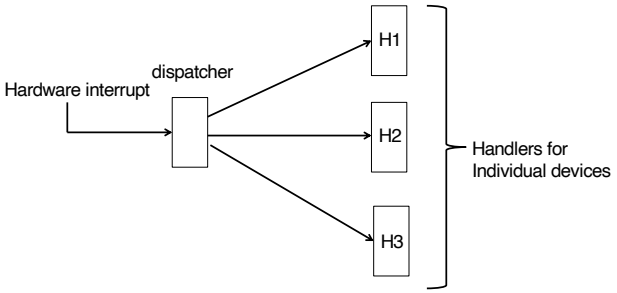
17

Interrupt Dispatching

- What is the division of labor between interrupt controller hardware and software?
- In software, how much is written in assembly and how much in a high-level language like C?
  - Obviously at least the register manipulation must be in assembly
- Differences in the BeagleBone and Galileo
  - The BB raises an IRQ exception and passes the IRQ number to the CPU
  - The Galileo interrupt controller holds the entire interrupt vector, calling the appropriate handler directly

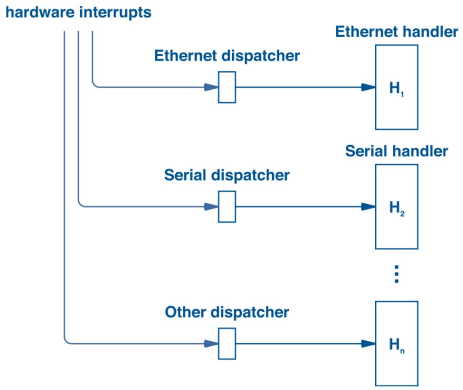
18

Dispatching to Handlers



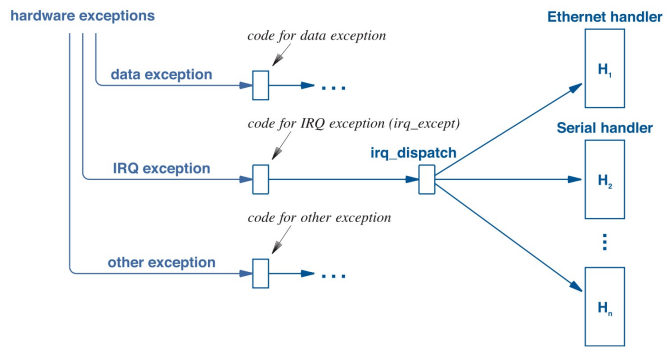
19

Galileo



20

BeagleBone



21

irq_dispatch

```
/*---------------------------------------------------------------------
 * irq_dispatch - call the handler for specific interrupt
 *---------------------------------------------------------------------
 */

void    irq_dispatch(){

    struct  intc_csreg *csrptr = (struct intc_csreg *)0x48200000;
    uint32  xnum;           /* Interrupt number of device   */
    interrupt (*handler)(); /* Pointer to handler function  */

    /* Get the interrupt number from the Interrupt controller */

    xnum = csrptr->sir_irq & 0x7F;

    /* If a handler is set for the interrupt, call it */

    if(intc_vector[xnum]) {
    handler = intc_vector[xnum];
    handler(xnum);        }

    /* Acknowledge the interrupt */

    csrptr->control |= (INTC_CONTROL_NEWIRQAGR);
}
```

22

Disabling Interrupts

- While an interrupt is being handled, further interrupts are disabled
- They remain disabled in the dispatcher and when the handler is called and returns
- Only re-enabled once the system is returning to user code
- This places constraints on how long the system can run with interrupts disabled

23

Other constraints

- The interrupt hander can never invoke a function that will block
  - signal() but not wait()
  - The Null process must be current or ready
- Scheduling during an interrupt may be required
  - Maintain the scheduling invariant

24

Exception Vectors (start.S)

```
/* ARM exception vector table.  This is copied to address 0.  See A2.6
 * "Exceptions" of the ARM Architecture Reference Manual.  */
_vectors:
    ldr pc, reset_addr      /* Reset handler */
    ldr pc, undef_addr   /* Undefined instruction handler */
    ldr pc, swi_addr     /* Software interrupt handler */
    ldr pc, prefetch_addr   /* Prefetch abort handler */
    ldr pc, abort_addr   /* Data abort handler */
    ldr pc, reserved_addr   /* Reserved */
    ldr pc, irq_addr     /* IRQ (Interrupt request) handler */
    ldr pc, fiq_addr     /* FIQ (Fast interrupt req) handler */

reset_addr:     .word reset_handler
undef_addr:     .word reset_handler
swi_addr:       .word reset_handler
prefetch_addr:  .word reset_handler
abort_addr:     .word reset_handler
reserved_addr:  .word reset_handler
irq_addr:       .word irq_handler
fiq_addr:       .word reset_handler

_endvectors:
```

25

end

26