Week 6. Software Interrupts

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Overview of the week

- Exception Handling on the ARM Processor
- What is a **S**oft**W**are **I**nterrupt (SWI)?
  - What are SWIs useful for?
  - What happens on an SWI?
  - What happens on SWI completion?
  - How to install a SWI handler
- Announcement:
  - Quiz on Thursday
  - Open books, open notes (no sharing of notes)
  - Bring a calculator (if you cannot do binary/hex arithmetic)
  - No laptops, cell phones or any other device that can be used to communicate with others
  - Useful tip: **understand** and print the 4-page ARM instruction summary

Optimization for Code Size – Optimizing Structures

- Which of the two structures would be better?

```
struct
{
        char a;
        int b;
        char c;
        short d;
}
```

12 bytes

```
struct
{
        char a;
        char c;
        short d;
        int b;
}
```

8 bytes

More Space Optimization

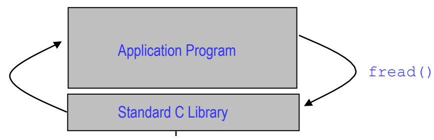
- Can use the `__packed` key word to instruct the compiler to remove all padding

```
__packed struct
{
        char a;
        int b;
        char c;
        short d;
}
```

8 bytes

- Packed structures are slow and inefficient to access
- ARM Compiler emulates unaligned load and store by using several aligned accesses and using several byte-by-byte operations to get the data
- Use `__packed` only if space is more important than speed and you cannot reduce padding by rearrangement

Reading Files from User Applications



System Calls

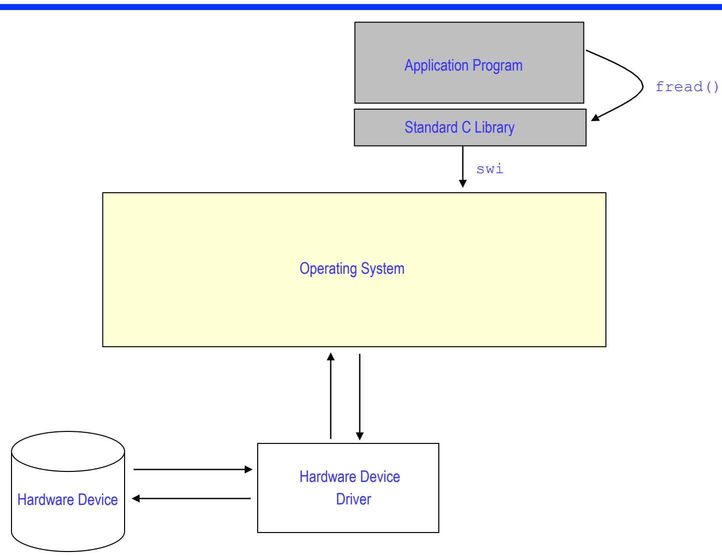
- Systems Calls are ways by which programs running in user mode request services from an operating system

- Programs running in user mode typically have restrictions on their ability to directly interact with system resources
  - Most systems do this to protect system resources from errant or malicious programs
  - Typical sequence by which user mode programs request resources from an OS maybe
    - Programs invoke system calls to request/interact with system resources (example: program may use the `open` system call to request the OS to open a file on its behalf)
    - OS verifies whether the resource can be safely allocated to the program or if the program has access to the requested resource (in the previous example, the OS verifies whether the user program making the `open` system call has read/write permissions to the file, the file exists…)
    - If the requesting program does not have access to the resource or a resource cannot be safely allocated to the program, an error is returned to the program invoking the system call otherwise OS performs the requested service and returns the result (in the previous example of `open` system call, the OS will return a file descriptor to the file)
- How are system calls represented differently from function calls?

What are SWIs?

- **S**oft**W**are **I**nterrupt (SWI) is an instruction provided by the ARM Instruction Set
- SWI instruction is used to generate an exception
  - Executes in the privileged mode
- SWI provides a way for a program running in User mode to request a service that can only be accessed in a privileged mode
  - Typically, used by applications to call operating system routines

SWI in Action



SWIs or Function Calls?

- SWIs are similar to a sub-routine call because
  - Parameters and return values can be passed through registers
- Differs from a sub-routine call because the ARM processor
  - Stashes away user mode CPSR
  - Switches to supervisor mode (if not already in supervisor mode)
  - Starts executing from a specific location (always 0x08)

Calling SWIs from C Code

User-Level C Source Code

```
char __swi(4) SWI_ReadC(void);
void __swi(6) SWI_WriteC(char);
void readline (char *buffer)
{
    char ch;
    do {
       *buffer++ = ch =
       SWI_ReadC();
    while (ch != 13);
   }
   *buffer = 0;
} /* end readline() */
```

Source : *Jumpstart Programming Techniques*

Assembly code produced by compiler

```
readline
        STMDF   sp!,{lr}
        MOV     lr, a1
readagain
        SWI     0x04
        STRB    a1,[lr],#1
        CMP     a1,#&d
        BNE     readagain
        MOV     a1,#0
        STRB    a1, [lr, #0]
        LDMIA   sp!, {pc}
```

Exception Handling

- Exception Handler
  - Most exceptions have an associated software exception handler that executes when that particular exception occurs
- Where is this exception handler located?
- Vector table
  - Reserved area of 32 bytes at the end of the memory map (starting at address 0x0)
  - One word of space for each exception type
  - Contains a Branch or Load PC instruction for the exception handler
- Exception modes and registers
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own r13 (stack pointer)
    - Its own r14 (link register)
    - Its own spsr (Saved Program Status Register)
  - Exception handlers must save (restore) other register on entry (exit)

Exception Handling

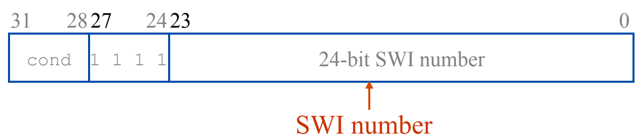
- When an exception occurs, the ARM:
  - Copies cpsr into spsr_<mode>
  - Sets appropriate cpsr bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the pc-4 in lr_<mode>
  - Sets pc to vector address
- To return, exception handler needs to:
  - Restore cpsr from spsr_<mode>
  - Restore pc from lr_<mode>

⋮	⋮
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

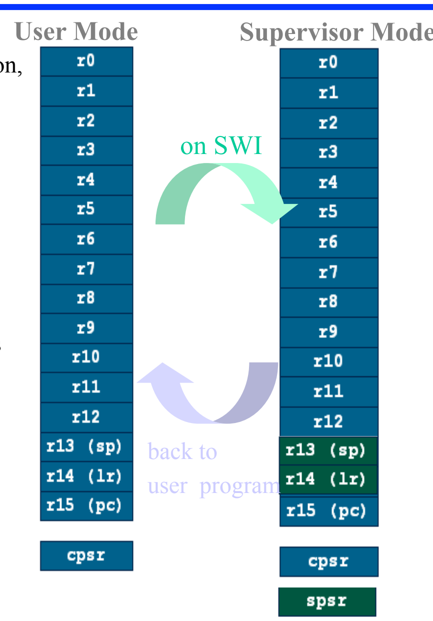
`SWI` Instruction Format

- Example: `SWI 0x18`
  - The number identifies the handler to be called
  - Example `0x18` could refer to a SWI handler that reads key input whereas `0x19` could refer to a SWI handler that outputs a char etc. …
- The last 24-bits of the SWI instruction contain the SWI number



SWI Handling

- Assume initially in User mode
- When the processor executes the SWI instruction, the **ARM processor**
  - Copies `cpsr` into `spsr_svc`
  - Sets appropriate `cpsr` mode bits to 10011 (`svc` mode)
    - Disables IRQs
  - Stores return address (`pc` – 4) in `lr_svc`
  - Sets `pc` to vector address 0x08
- To return, **exception handler** needs to:
  - Restore `cpsr` from `spsr_svc` (`cpsr` now has mode bits 10000 = `usr` mode)
  - Restore `pc` from `lr_svc`
- In privileged modes, a single instruction can be used to cause the SPSR for the current mode to be copied into CPSR while copying `lr` into `pc`
  - `MOVS pc, lr`

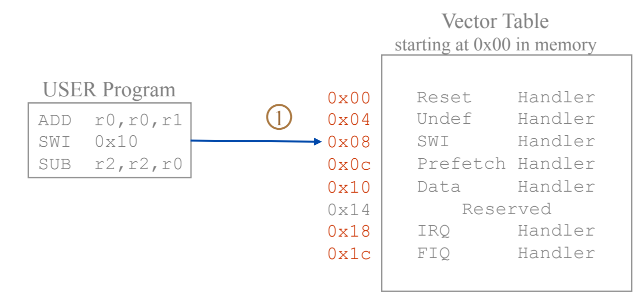


What Happens on a `SWI`? (1)

- The ARM architecture defines a **Vector Table** indexed by exception type
- On `SWI`

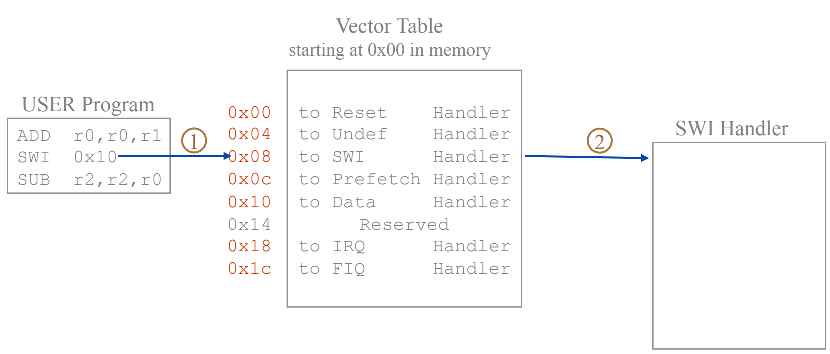
```
lr_svc             ⇦ pc-4
spsr_svc           ⇦ cpsr
cpsr(mode bits)    ⇦ 10011
pc                 ⇦ 0x08
```

(supervisor mode, no IRQ)
(address of SWI vector)



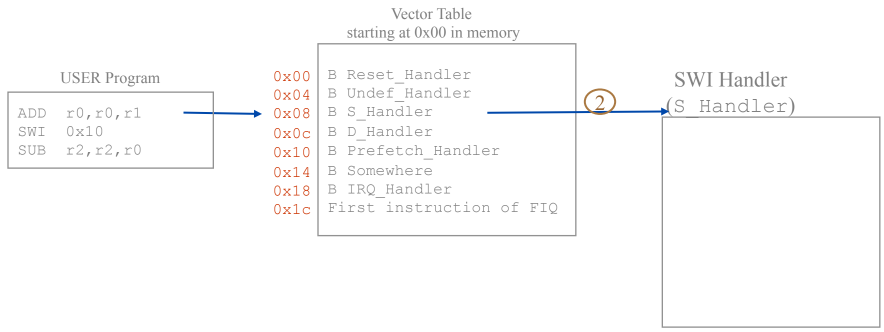
What Happens on a `SWI`? (2)

- Not enough space in the table (only one instruction per entry) to hold all of the code for the `SWI` handler function
- This *one* instruction must transfer control to appropriate SWI Handler



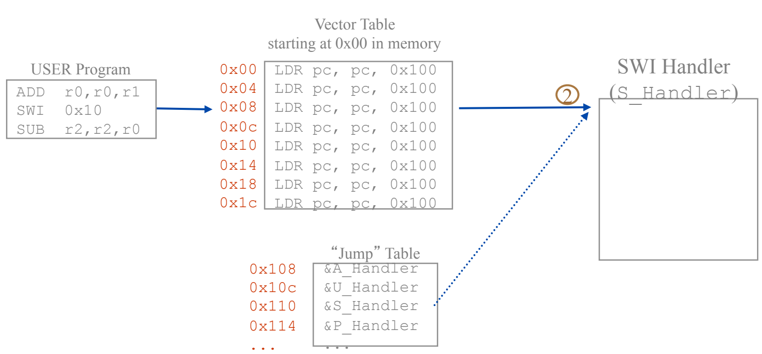
“Vectoring” Exceptions to Handlers

- **One option**: Use a branch instruction (limited range because branch offset must be within 32 MB of branch instruction)



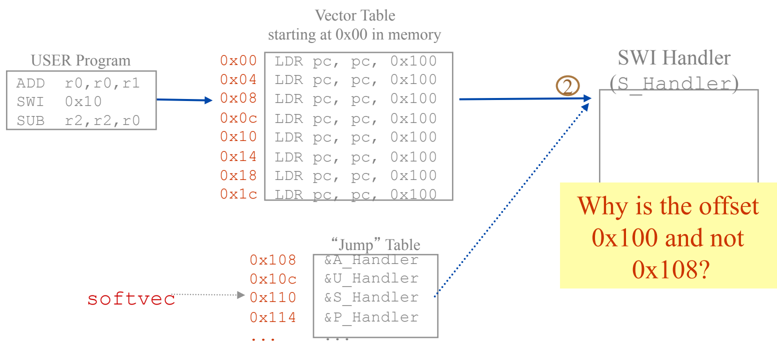
“Vectoring” Exceptions to Handlers

- **Option of choice**: Load the address of the actual handler into the `pc` using `LDR` instruction
  - Store the address of the SWI handler (`S_Handler`) in a memory location
  - Use `LDR` instruction to load `pc` with **contents** of this memory location



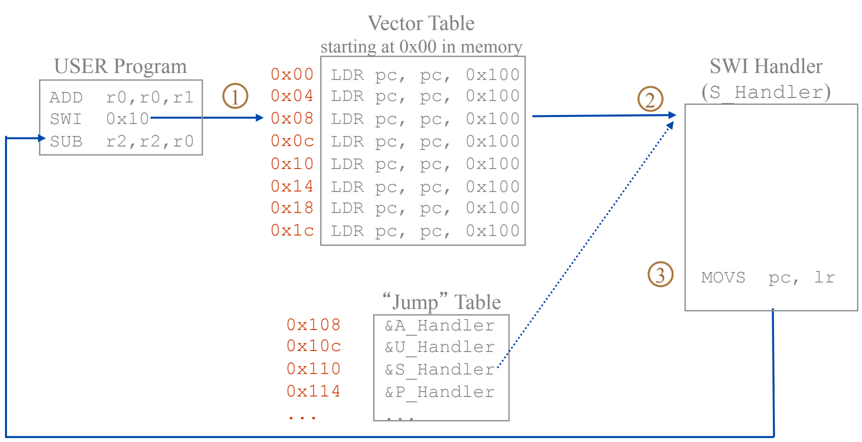
“Vectoring” Exceptions to Handlers

- **Option of choice**: Load the address of the actual handler into the `pc` using `LDR` instruction
  - A memory location (call it `softvec`) stores the address of the SWI handler (`S_Handler`)
  - Use `LDR` instruction to load `pc` with **contents** of `softvec`
    - Compute the offset `(#offset)` of `softvec` from SWI vector and use the `LDR pc, [pc, #offset]` to load the address of `S_Handler` into `pc`
  - Offset for LDR instruction is represented by 12 bits; hence `softvec` should be a max of 4K away from 0x08 , but address stored in `softvec` can be a 32-bit address



What Happens on `SWI` Completion?

- When the handler is done, it returns to the USER program -- at the instruction following the `SWI`
- `MOVS` restores the original `CPSR` as well as changing `pc`



`SWI` Instruction Format Again!

- ARM processor ignores the last 24-bits of the SWI instruction (containing the SWI number)

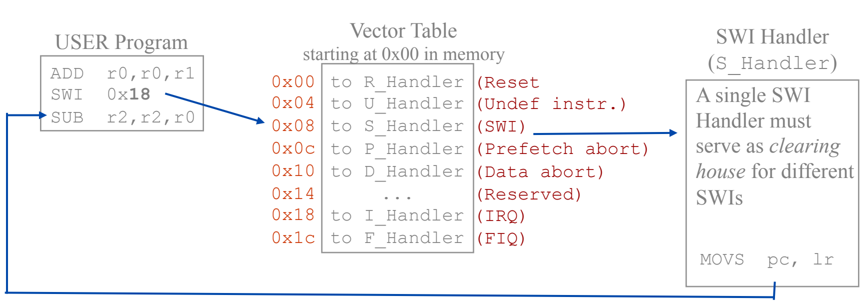
31 28	27 24	23 0
cond	1 1 1 1	24-bit “comment” field (ignored by processor)

SWI number

- Exercise – how would we extract the SWI number, if we knew that the 32-bit representation of the instruction (`SWI 0x18`) was stored in a register (say `r0`)?
  - We will discuss how to get the instruction encoding into `r0` later
- If `r0` stored the 32-bit representation of the instruction (`SWI 0x18`) how would you extract the SWI number
  - You want the last 24 bits of the instruction
  - Use bit-clearing (`BIC`) to zero out the 8 MSBs
  - `BIC r0, r0, #0xff000000;` `r0` ⇦ `r0 AND NOT(0xff000000)`
  - This effectively sets `r0` to the result of (`r0` AND `~0xff000000`)

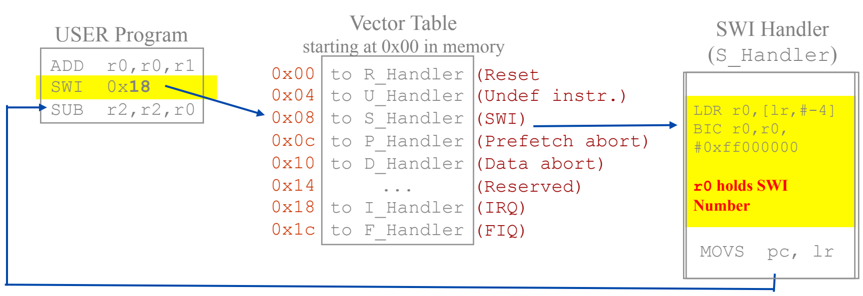
Determining the SWI number – I

- **Problem**: <u>All</u> `SWI`s go to 0x08 regardless of the number that you put after the SWI instruction
  - What’s the point of that number, then?
  - How do you actually find out which SWI to execute if they all end up at the same address?
- When you are at `S_Handler` in `svc` mode
  - Is it possible for you to find out where you came from in `usr` mode?
  - Can you exploit this information to find out which SWI you should execute?



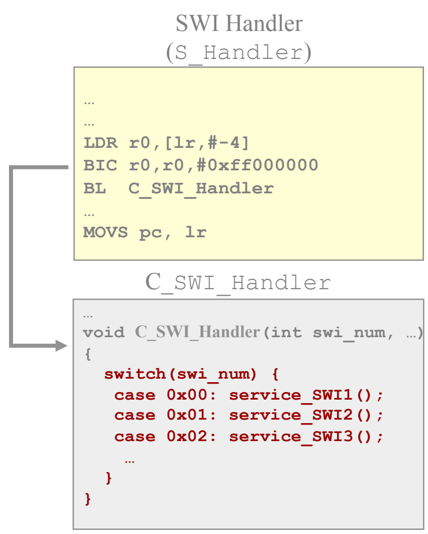
Determining the SWI number – II

- Take a look at `lr_svc`
  - Holds information about the instruction in user space that immediately follows the SWI (done by the processor at the time that it switches into `svc` mode)
  - If you step back 4 bytes from `lr_svc`, what do you get?
    - `LDR r0,[lr,#-4]`
  - `r0` stores the 32-bit encoding of the SWI instruction, in this case, SWI 0x18
    - Last 24 bits ("comment field) contains the SWI number
  - Extract the SWI number now by clearing out everything but the last 24 bits
    - `BIC r0,r0,#0xff000000`
- Done! We now have the SWI number in `r0`



Use The `SWI` # to Jump to "Service Routine"

- Key Insight – need access to general purpose register (`lr` in this case)
  - Initial part of the SWI Handler has to be written in assembly
  - Exit also in assembly
- Once we capture the SWI number and store it in `r0`
  - Call a C function from the assembly code to service the specific SWI
  - Easier to write SWI handler in C rather than assembly
- Remember ATPCS conventions
  - `r0` contains the first argument of a function call
- **Caution**:
  - In `S_Handler` the address of the return instruction (to user program) is stored in `lr_svc`
  - `BL` call will overwrite the value of `lr_svc`
  - Need to save `lr_svc` in `S_Handler` before using `BL`

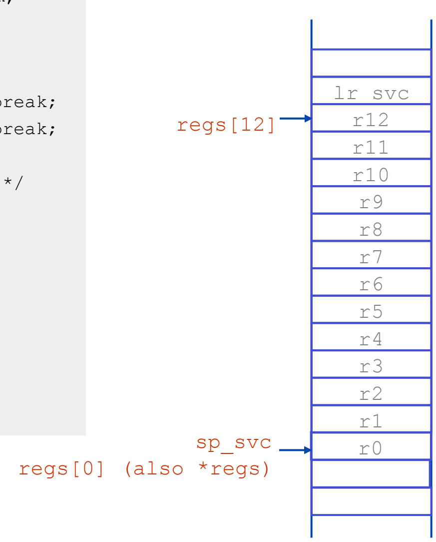


SWI Handler

```
S_Handler
    STMFD   sp!, {r0-r12, lr} ;store user's gp registers and lr_svc
    MOV     r1, sp  ;r1 now contains pointer to parameters on stack
    LDR     r0, [lr, #-4]       ; extract the SWI number
    BIC     r0,r0,#0xff000000 ; get SWI # in r0 by bit-masking
    BL      C_SWI_handler       ; go to handler (see next slide)
    LDMFD   sp!, {r0-r12, lr} ; unstack user's registers and lr_svc
    MOVS    pc, lr              ; return from handler
```

C_SWI_Handler

```
void C_SWI_handler(unsigned swi_num,
   unsigned *regs)
{
    switch (swi_num) {
    case 0: /* SWI number 0 code */ break;
    case 1: /* SWI number 1 code */ break;
    ...
    case XXX: /* SWI number XXX code */
       break;
    default:
    } /* end switch */
} /* end C_SWI_handler() */
```

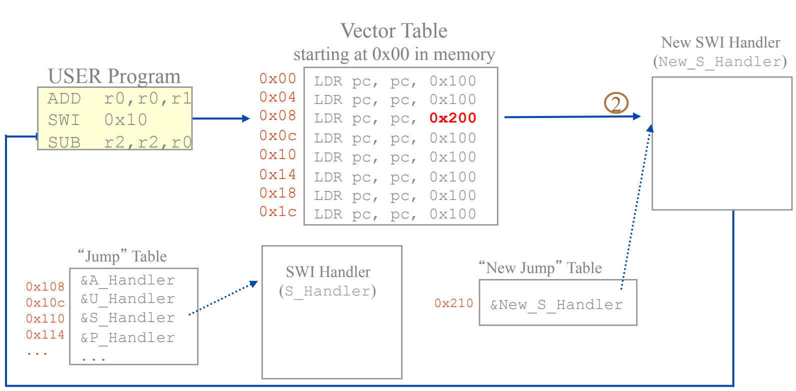


Adding your custom SWIs

- Suppose you are given an OS with some of the standard syscalls already defined
- You want to add a few more custom syscalls that you want to be able to provide your application developers
- How do we add our custom SWIs without losing the SWI handling already provided?
  - Modify the code for the `C_SWI_Handler` if you have the source code
  - Wire in your SWI Handler before the system SWI handler

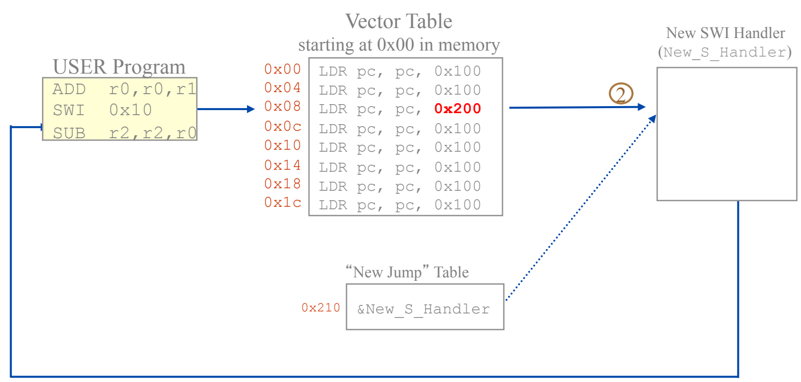
Installing your SWI handler

- Two ways to install your own SWI handler
- Method 1
  - Change the address of `S_Handler` (stored in `softvec)` to point to the new SWI handler
- Method 2 (done in lecture)
  - Define a new location for `softvec`
  - Store the address of the new SWI handler in `softvec`
  - Change the instruction stored in vector table location 0x08 to point to the new `softvec`



Installing the SWI handler

- Load `pc` method
  - Compute the offset to be used in the `LDR` instruction (at 0x08)
    - Take the address of the word containing the address of the new SWI handler
    - Subtract the address of the SWI exception vector in the vector table
    - Subtract 8 bytes
    - Check that the result can be represented in 12 bits
  - Logically OR this with 0xe59FF000 (the opcode for `LDR pc, [pc, #0]`)
  - Store this new `LDR` instruction in the SWI exception vector



Loading the Vector Table

```
unsigned Install_Handler(unsigned location, unsigned int *vector)
/*Updates the contents of 'vector' stored at 0x08 to contain LDR pc, [pc, #offset] instruction to cause
    long branch to              'location' */
/*Function returns the original contents of 'vector' */
{
    unsigned offset;
    unsigned vec, oldvec;
    offset = ((unsigned) location - (unsigned) vector - 0x8)
    if(offset & 0xFFFFF000)  /* check if the offset can be represented using 12 bits */
    {      printf("Installation of handler failed");
           exit(0); }
    vec =(offset|0xe59FF000);/* vec now contains LDR pc, [pc, #offset] */
    oldvec = *vector;        /* sa                          08 */
    *vector = vec; /* replace the contents of 0x08 with the new LDR instruction */
    return(oldvec);                           ction at 0x08 for chaining */
}
```

location = 0x210

vector = 0x08

*vector = 0xe59ff100

vec = 0xe59ff200

*vector = 0xe59ff200

oldvec = 0xe59ff100

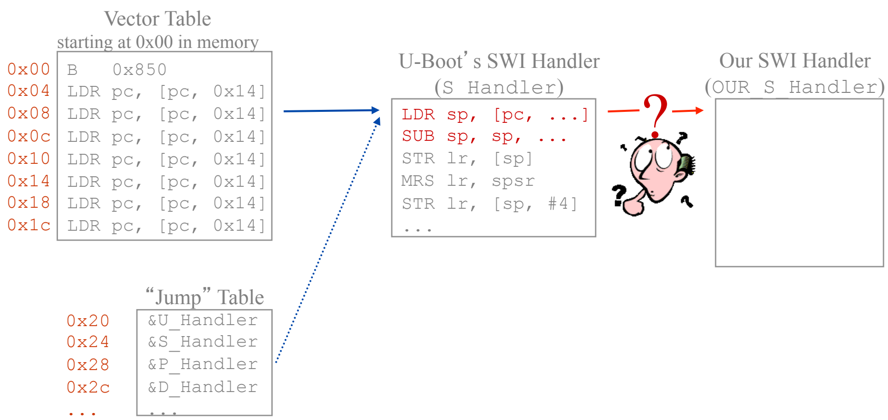
The following code calls function

```
unsigned *vector= (unsigned *) 0x08;
unsigned *location= (unsigned *) 0x210; /*The address of the new softvec */

Install_Handler((unsigned) location, vector);



*location= (unsigned) New_S_Handler;//Store the address of your New_S_Handler in swiaddr
```

Source : *Jumpstart Programming Techniques & ADS Developer's Guide*

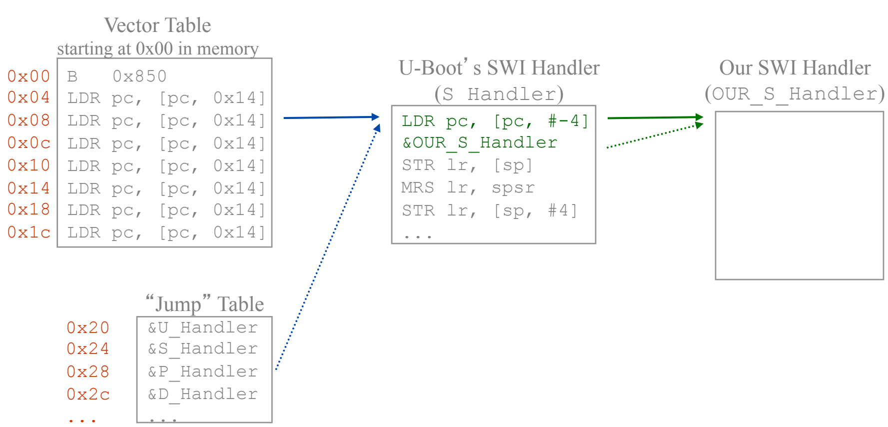
Installing the SWI Handler – Our Way

- **Problem**: Can't write to either the Vector Table or the "Jump" Table (both in ROM)
- **Solution**: Hijack the first few instructions of U-Boot's SWI Handler
  - Redirect the processor to our own handler
  - Only eight bytes guaranteed to be available (arbitrary lab restriction)



Installing the SWI Handler – Our Way

- Which instructions to use?
  - Analogous situation to the SWI vector in the Vector Table
  - One instruction must transfer control to Our SWI Handler
  - Eight bytes available → four bytes for instruction, four bytes for address



Reentrant Exception Handlers

- Reentrant programming – used to describe code which can have concurrent invocations which should ideally not interfere with each other

- What happens if you want to call another SWI from your SWI handler?
  - When an SWI occurs, the ARM
    - Sets appropriate CPSR mode bits to 10011 (`svc` mode)
    - Copies CPSR into `spsr_svc`
    - Stores return address (`pc` – 4) in `lr_svc`
    - Sets `pc` to vector address 0x08

- Need to save certain registers (that do not need to be saved otherwise)

- Do you see any issues here?

Program Status Register Instructions

- Two instructions to directly control a Program Status Register
- MRS
  - Transfers the contents of either the `cpsr` or the `spsr` into a register
- MSR
  - Transfers the contents of a register into the `cpsr` or the `spsr`

```
PRE    cpsr = nzcvqIFt_SVC

   MRS r1, cpsr
   BIC  r1, r1, #0x080
   MSR cpsr, r1

POST   cpsr = nzcvqiFt_SVC
```

A Reentrant `SWI` Handler for Lab2

```
import C_SWI_Handler
export S_Handler
S_Handler
    SUB     sp, sp, #4           ; leave room on stack for SPSR
    STMFD   sp!, {r0-r12, lr} ; store user's gp registers
    MRS     r2, spsr             ; get SPSR into gp registers
    STR     r2, [sp, #14*4]      ; store SPSR above gp registers
    MOV     r1, sp               ; pointer to parameters on stack
    LDR     r0, [lr, #-4]        ; extract the SWI number
    BIC     r0,r0,#0xff000000 ; get SWI # by bit-masking
    BL      C_SWI_Handler        ; goto handler (see prev lecture)
    LDR     r2, [sp, #14*4]      ; restore SPSR (NOT "sp!")
    MSR     spsr, r2             ; restore SPSR from r2
    LDMFD   sp!, {r0-r12, lr} ; unstack user's registers
    ADD     sp, sp, #4         ; remove space used to store SPSR
    MOVS    pc, lr               ; return from handler
```

Source : *Jumpstart Programming Techniques*