ARM Procedure Call Standard

(Subtitle : The Implementation of backtrace function)

2008. 2. 22

LG Electronics / Software & Solution 센터
S/W D Gr. 서희 (seohee@lge.com)

Introduction

- Assumption
  - Wrong arguments of function cause segmentation fault
  - It is very popular function and is called by many functions!
- Trouble
  - Which of the functions does the problem come from?
- How to get start?
  - In the time of executing segmentation fault handler, the stack has a lot of information for debugging such as return address of functions, CPU context and so on.
  - If we get return address of functions from stack, we can see function's call history called as 'backtrace information'.
  - __builtin_return_address

__builtin_return_address usage

- `__builtin_return_address (LEVEL)
  - Getting the Return of a Function
  - This function returns the return address of the current function, or of one of its callers. The LEVEL argument is number of frames to scan up the call stack.

- Trouble
  - Sometimes __builtin_return_address function is not working or can get the only information of level ‘0’

APCS (Arm Procedure Call Standard) [1/8]

- The APCS defines:
  - restrictions on the use of registers
  - conventions for using the stack
  - passing/returning arguments between function calls
  - the format of a stack-based structure which may be 'backtraced' to provide a list of functions (and parameters given) from the failure point backwards to the program entry
  - Compiler option : -apcs

APCS (Arm Procedure Call Standard) [2/8]

	APCS	APCS Role	reg	APCS	APCS Role
0	a1	argument 1 integer result	8	v5	register variable 5
1	a2	argument 2	9	sb/v6	Static base / register variable 6
2	a3	argument 3	10	sl/v7	stack limit / register variable 7
3	a4	argument 4	11	fp	frame pointer
4	v1	register variable 1	12	ip	scratch reg. / new sb in inter-link-unit calls
5	v2	register variable 2	13	sp	Lower end of current stack frame
6	v3	register variable 3	14	lr	link address
7	v4	register variable 4	15	pc	program counter

APCS (Arm Procedure Call Standard) [3/8]

- Both the ARM and Thumb instruction sets contain a primitive subroutine call instruction, BL, which performs a branch-with-link operation.

- Subroutine call
  - For example in ARM-state, to call a subroutine addressed by r4 with control returning to the following instruction, do:

```
MOV LR, PC  ( PC register value will be saved into LR register. )
BX r4       ( Jump subroutine )
```

- Subroutine return

```
MOV PC, LR  ( restore LR register value to PC )
```

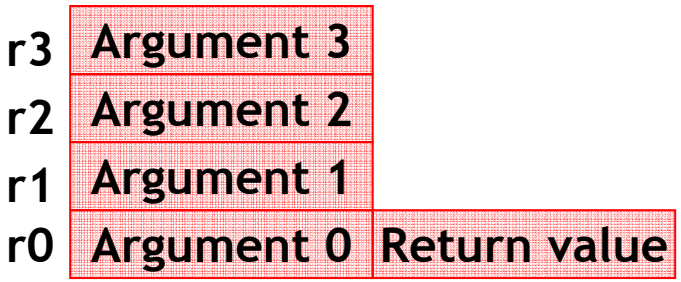
APCS (Arm Procedure Call Standard) [4/8]

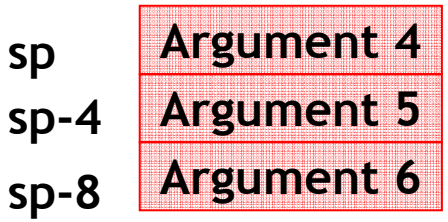
❑ Parameter Passing

- ➲ Passing arguments: core registers (r0-r3) and on the stack
- ➲ For subroutines that take a small number of parameters, only registers are used.
- ➲ Passing argument for long long type: pair of consecutive argument registers (e.g., r0 and r1)

❑ Return value

- ➲ Integer or pointer: r0
- ➲ Two-word: r0 and r1





APCS (Arm Procedure Call Standard) [5/8]

- **Stack**
  - **Linked list of 'frames' which are linked through what is known as a 'backtrace structure‘**
  - **Stored at the high end of each frame**
  - **Allocated in descending address order**
  - **The register sp**
    - Point to the lowest used address in the most recent frame.

APCS (Arm Procedure Call Standard) [6/8]

❑ Backtrace

- ➲ The register fp (frame pointer) should be zero, or it should point to the last in a list of stack backtrace structures which will provide a means of 'unwinding' the program to trace backwards through the functions called

```
save code pointer [fp] ◂ – – – – fp points here
return ip value [fp, #-4]
return link value [fp, #-8]
return pc value [fp, #-12]
return fp value [fp, #-16] points to next structure
[saved other registers]
```

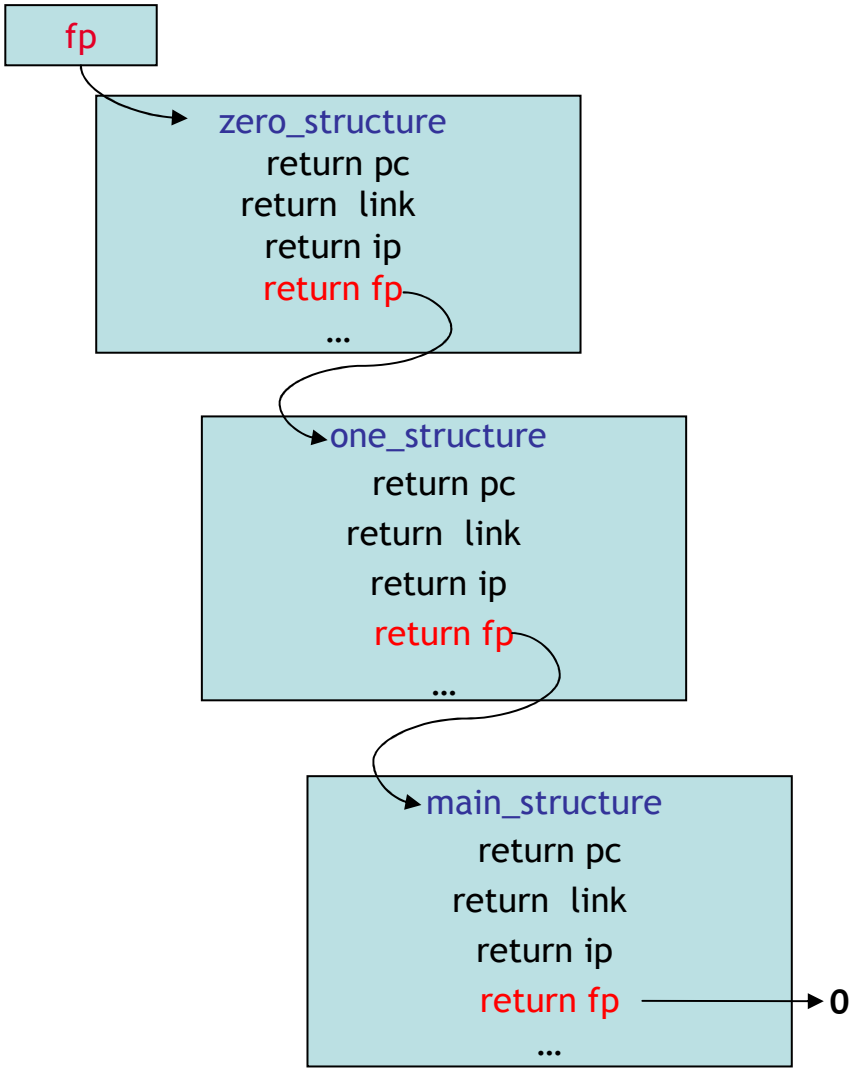
APCS (Arm Procedure Call Standard) [7/8]

```
int main(void)
{
   one();
   return 0;
}

void one(void)
{
   zero();
   two();
   return;
}

void two(void)
{
   printf("main...one...two\n");
   return;
}

void zero(void)
{
   return;
}
```

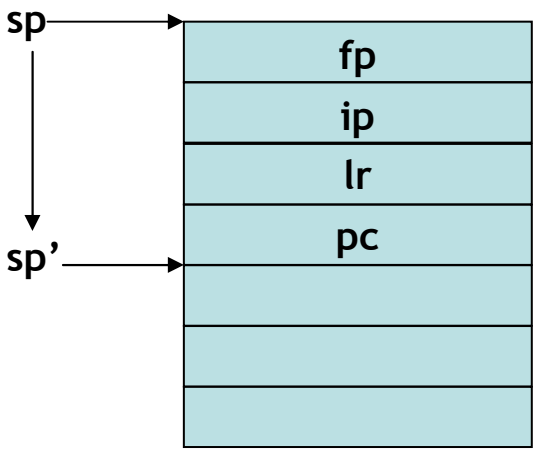


APCS (Arm Procedure Call Standard) [8/8]

```
main:
   mov        ip, sp
   stmfd      sp!, {fp, ip, lr, pc}
   sub        fp, ip, #4
   bl         one
   mov        r0, #0
   b          .L2

one:
  mov         ip, sp
  stmfd       sp!, {fp, ip, lr, pc}
  sub         fp, ip, #4
  bl          zero
  bl          two
  b           .L3

two:
  mov         ip, sp
  stmfd       sp!, {fp, ip, lr, pc}
  sub         fp, ip, #4
  ldr         r0, .L5
  bl          printf
  b           .L4
zero:
  mov         ip, sp
  stmfd       sp!, {fp, ip, lr, pc}
  sub         fp, ip, #4
  b           .L7
```



stmfd sp!, {fp, ip, lr, pc}

Implementation of simple backtrace function [1/2]

❑ Key concept

- ➲ The fp register points to the stack backtrace structure for the currently executing function
- ➲ The return fp value should be zero, or a pointer to a stack backtrace structure created by the function which called the current function
- ➲ The return fp value in this structure is a pointer to the stack backtrace structure for the function that called the function that called the current function; and so on back until the first function

❑ Proto-type

- ➲ my_bt_return_address(unsigned short bt_level)

❑ Function

- ➲ This function returns the return address of the current function, or of one of its callers. The LEVEL argument is number of frames to scan up the call stack.
- ➲ A value of `0' yields the return address of the current function, a value of `1' yields the return address of the caller of the current function, and so forth.

Implementation of simple backtrace function [2/2]

- Sample source code

```
void my_bt_retrun_address(unsigned short bt_level)
{
        unsigned int *fp, i;

        printf("Backtrace: fp=%x", (int) fp);
        for (i = 0; i < bt_level; i++) {
                if ((i % 8) == 0)
                        printf("\n");

                printf(" %08x",         );

        }
        printf("\n");
}
```

Performance or Debugging ? [Trade Off]

- omit-frame-pointer
  - omit-frame-pointer option allows fp register to use as common register for performanceNot using frame-pointer as debugging register, we can't get backtrace information from the system.
- no-omit-frame-pointer
  - For debugging

Conclusion

- Gcc compile provide backtrace structure to programmers
- The fp register points to the stack backtrace structure for the currently executing function
- The return fp value in this structure is a pointer to the stack backtrace structure for the function that called the function that called the current function; and so on back until the first function
- In the case of mips machine, it doesn't use frame pointer register for debugging. That's why mentioned backtrace function unfortunately will be not working.