Universidade Federal de Santa Catarina

Sistemas Operacionais II

ARM Cortex-A9: Initialization + Bootstrapping

Caio Pereira Oliveira

Ricardo do Nascimento Boing

Thomas Fernandes Feijoo

1. Introdução

O objetivo deste trabalho é apresentar o processo de inicialização e bootstraping para o processador ARM Cortex-A9 e responder as seguintes perguntas:

- Como ocorre o boot no ARM Cortex-A9
- Como criar uma imagem bootavel para o QEMU?
- Como programar em C++ para o ARM Cortex-A9?

Neste capítulo iremos citar os componentes básicos para executar código *bare metal* para ARM Cortex-A9.

1.1 Linux

As ferramentas utilizadas são feitas primariamente para Linux, então escolhemos utilizar a distribuição Ubuntu.

1.2 GCC Cross-Compiler Toolchain

Não é possível utilizar o compilador GCC do sistema para compilar o código que irá executar em um sistema ARM, por isso precisamos de um cross-compiler.

Cross-compiler é um compilador que é capaz de produzir código executável para uma plataforma diferente da qual o compilador está sendo executado. A GNU build tools, usa o conceito de *target triplet* para descrever uma plataforma. Essa tripla consiste de: a arquitetura da plataforma, o vendor e o sistema operacional ou o tipo de interface binária.

Para este seminário, precisamos selecionar a toolchain correta, ou seja, o cross-compiler toolchain que contenha um *target triplet* correspondente ao nosso objetivo. A utilizada foi adquirida na documentação do EPOS, mais especificamente a versão *GCC-7.2.0* para ARMv7.

1.3 QEMU

Para emular uma máquina ARM, utilizamos o QEMU, um *hypervisor* que permite uma virtualização de um sistema computacional completo.

Como o código eventualmente irá rodar em um dispositivo ARM real, é mais fácil começarmos utilizando um emulador, por motivos de:

- Não é necessário hardware adicional, ou seja, pode ser executado no computador pessoal do desenvolvedor.
- Melhores ferramentas para debuggar o estado do hardware emulado.

- Aumento de velocidade

Como o QEMU suporta uma grande variedade de sistemas, precisamos instalar a versão ARM. Em sistemas Debian/Ubuntu, o pacote *qemu-system-arm* provê o que necessitamos.

Após isso, escolhemos a máquina ARM a ser emulada, para este seminário a máquina escolhida foi a *realview-pbx-a9*

1.4 Build system

Precisamos dos componentes essenciais para construir um sistema. Utilizamos a *build tool Make*, e o pacote *build-essential,* presente em sistemas baseados em Debian, para instalar o *Make* e outros programas relevantes

2. Programa de Teste

O objetivo do programa de teste é demonstrar a inicialização no Cortex-A9 e sua configuração para executar código C++.

Iremos demonstrar como construtores e destrutores globais são invocados e como a seção bss é zerada antes do programa iniciar.

Para demonstrar o funcionamento, escreveremos as informações na porta serial UART.

2.1 UART

UARTs (universal asynchronous receiver-transmitter) são dispositivos de *hardware* que possibilitam a comunicação (transmissão e recepção) de dados na forma serial [3]. Para trabalhar com a UART é necessário acessá-la através do seu endereço de memória. A UART0, que será utilizada no programa de teste, está localizada no endereço 0x10009000.

3. Startup

O arquivo startup.s é onde a entrada do programa acontece.

```
.section .vector_table
.global _reset
_reset:
b _start // 0x0 Reset
b .      // 0x4 Undefined Instruction
b .      // 0x8 Software Interrupt
b .      // 0xC Prefetch Abort
b .      // 0x10 Data Abort
b .      // 0x14 Reserved
b .      // 0x18 IRQ
b .      // 0x1C FIQ
```

No início do arquivo, definimos a seção .vector_table. Cada entrada da tabela corresponde a um tratador de diferentes tipos de interrupção.

Para o programa de testes, estamos apenas interessados no Reset, todas as outras interrupções levam a um loop infinito.

```
.section .entry
_start:
// init stack
ldr sp, =_stack_end
```

Logo após, definimos a seção entry e o símbolo _start, que é o ponto de entrada do programa.

Também inicializamos o *stack pointer* com o endereço fornecido pelo *linker*. Caso não tivéssemos inicializado o *stack pointer*, não seria possível chamar funções. A convenção para a pilha é que ela deve crescer em direção ao endereço de memória mais baixo, logo, o fundo da pilha é a posição de memória mais alta (_stack_end).

```
// clear bss
mov r0, #0
ldr r1, =_bss_start
ldr r2, =_bss_end

bss_loop:
cmp   r1, r2
strlt r0, [r1], #4
blt   bss_loop
```

Agora devemos zerar a área da memória correspondente a seção bss, para ter certeza que o programa em C++ lerá os valores corretos dos objetos globais.

```
// init static objects
ldr r0, =_init_array_start
ldr r1, =_init_array_end

globals_init_loop:
cmp   r0, r1
ldrlt r2, [r0], #4
blxlt r2
blt   globals_init_loop
```

Devemos então invocar todos os construtores globais do programa. A seção init_data é gerada pelo gcc, devemos chamar as funções registradas nela para que os objetos sejam inicializados.

```
// jump to main
bl main
```

Após isso, o programa está inicializado e podemos executar a função main, escrita em C++.

```
// destroy static objects in reverse order
ldr r0, =_fini_array_start
ldr r1, =_fini_array_end

globals_fini_loop:
cmp   r0, r1
ldrlt r2, [r0], #4
blxlt r2
blt   globals_fini_loop
```

Quando a função main terminar de executar, devemos então chamar todos os destrutores dos objetos globais na ordem inversa que foram inicializados.

4. Main

```
volatile unsigned int* const UART0DR = (unsigned int*) 0x10009000;

void print_uart0(const char *s) {
    while (*s != '\0') { /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
        s++; /* Next char */
    }
}
```

Função de escrita na UART0. Escreve sequencialmente cada caractere da string na posição de memória correspondente a UART0 até encontrar um null terminator.

```
class SideEffect {
public:
    SideEffect(const char* s): _s(s) {
        print_uart0("SideEffect created ");
        print_uart0(_s);
        print_uart0("\n");
    }

    ~SideEffect() {
        print_uart0("SideEffect destroyed ");
        print_uart0(_s);
        print_uart0("\n");
    }
private:
    const char* _s;
};

SideEffect se_global1("global 1");
SideEffect se_global2("global 2");
SideEffect se_global3("global 3");
SideEffect se_global4("global 4");
SideEffect se_global5("global 5");
```

Classe para demonstrar o funcionamento dos construtores e destrutores de objetos globais.

Cada objeto imprimirá uma string única que permite demonstrar a ordem da inicialização e destruição dos objetos globais.

```
int zero_array[] = {0, 0, 0, 0, 0, 0};

void test_bss_is_zero() {
    bool is_zero = true;

    for (int i = 0; i < sizeof(zero_array) / sizeof(zero_array[0]); ++i) {
        if (zero_array[i] != 0) {
            is_zero = false;
        }
    }

    if (is_zero) {
        print_uart0("zero_array was initialized\n");
    } else {
        print_uart0("zero_array was NOT initialized!!!\n");
    }
}
```

Array de zeros e função de teste para demonstrar o funcionamento do código que zera a memória da seção bss.

```
int main() {
    SideEffect se_local("local");

    test_bss_is_zero();

    print_uart0("Hello world!\n");

    return 0;
}
```

Função main. Objeto para demonstrar o funcionamento de construtores e destrutores locais.

5. Linker Script

A função do linker script é descrever como as seções são mapeadas dos arquivos de entrada para as seções na memória: bss, text, data, etc.

```
SECTIONS {
    . = 0x0;

    .startup : {
        src/startup.o(.vector_table)
    }
```

Na posição de memória 0x0 é colocada a seção .vector_table, com os ponteiros para cada tratador de interrupções.

```
. = 0x10000;

.text : {
    *(.entry)
    *(.text)
    *(.rodata)
}
```

Na posição de memória 0x10000 é colocada a seção .entry, que é a entrada do programa, seguida das seções .text, que contém o resto do código, e a seção .rodata, que contém dados apenas para leitura.

```
.bss : {
    _bss_start = .;
    *(.bss)
    . = ALIGN(8);
    _bss_end = .;
}

.data : {
    _data_start = .;
    *(.data)
    . = ALIGN(8);
    _data_end = .;
}

.init_array : {
    _init_array_start = .;
    *(.init_array)
    *(.init_array.*)
    _init_array_end = .;
}

.fini_array : {
    _fini_array_start = .;
    *(.fini_array)
    *(.fini_array.*)
    _fini_array_end = .;
}
```

Logo após são colocadas as seções .bss, .data, .init_array e .fini_array. Cada uma as seções possui símbolos que marcam seu início e fim para serem referenciadas em startup.s.

```
    . = ALIGN(8);
    _stack_start = .;
    . = . + 0x1000; /* 4kB of stack memory */
    _stack_end = .;
}
```

Por último, é reservada uma área na memória para a pilha de execução.

6. Compilação e Execução

Para compilar o código é necessário executar os seguintes passos:

```
arm-gcc -c -mcpu=cortex-a9 -g -fno-exceptions
-fno-threadsafe-statics -fno-use-cxa-atexit -nostdlib -lgcc
src/main.cpp -o src/main.o
```

Compila o arquivo main.cpp, gerando o arquivo objeto main.o. As flags servem para desabilitar funcionalidades padrões do GCC que não estão disponíveis no *bare metal.*

```
arm-as src/startup.s -o src/startup.o
```

Monta o arquivo startup.s, gerando o arquivo objeto startup.o

```
arm-ld -T main.ld src/*.o -o main.elf
```

Linka os arquivos objeto startup.o e main.o utilizando o linker script para gerar o arquivo ELF main.elf.

```
arm-objcopy -O binary main.elf main.bin
```

Converte o arquivo ELF main.elf para formato binário para que o QEMU possa executá-lo.

Para executar o arquivo main.bin no QEMU, deve se usar o seguinte comando:

```
qemu-system-arm -M realview-pbx-a9 -m 128M -nographic
-no-reboot -serial stdio -monitor telnet:0.0.0.0:1234,server,nowait
-kernel main.bin
```

As flags configuram o QEMU para utilizar o realview-pbx-a9. A flags -monitor em especial habilita um servidor telnet que permite inspecionar QEMU durante sua execução.

Todo o processo de compilação e execução foi automatizada em uma Makefile inclusa junto com o código.

Referências

[1] http://bravegnu.org/gnu-eprog/lds.html
[2] https://sourceware.org/binutils/docs/ld/Scripts.html
[3] http://newtoncbraga.com.br/index.php/telecom-artigos/1709-
[4] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0447j/Bbabegge.html
[5] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0440b/Bbabegge.html
[6] http://umanovskis.se/files/arm-baremetal-ebook.pdf
[7] https://arobenko.gitbooks.io/bare_metal_cpp/content/
[8] https://wiki.qemu.org/Documentation/Platforms/ARM
[9] https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/