嵌入式系统仿真实验第五讲

1 计算机的异常向量机制

前面我们实际一直只是在内存中仿真运行 ARM 程序，并没有涉及这是一个什么系统的概念，都是假设程序都是从 main 函数或者汇编的_start 全局入口开始的，没有关心计算机加电后到底是如何工作的机制。

一般来说，计算机被设计成上电复位后从内存的 0x0 开始处开始，按顺序由取指机构（由 PC 寄存器指定取指地址）取指（fetch），然后送入指令译码器进行译码（decode），按照指令类型，分别取出指令码、操作数，然后不同的寻址方式把数据从寄存器或者内存中送入执行部件进行执行（execute），PC 指针再向下移动，执行下一条指令。这个过程的讲解在我的讲义第二章，可以翻回去再看看。

以前我们用 X86 讲微机原理的时候，这部分属于 bios 程序的内容，一般不讲一个 BIOS 程序是如何设计出来的，因为 PC 机出厂的时候主板上的 BIOS 程序是固化好的，我们只会简单介绍 BIOS 和 DOS 系统功能调用有哪些而已。而嵌入式系统一般没有固化的 BIOS 程序，需要开发人员自行设计或者移植，比如 Lilo、grab、Blob、redboot、uboot 之类的 bootloader 程序，由它完成系统上电后的初始化以及操作系统引导。类似于 PC 机的 BIOS 程序。

我们可以把一般计算机的执行方式看成是一种类似于**向量中断**算法的机制，或者称为异常向量机制，也就是当计算机出现某种异常改变时（这个改变可能是软件指令引起，也可能是硬件引起的），如果这个中断响应是打开(enable)的话，计算机会自动处理这种异常的处理，处理的方式就是将 PC 自动指向这个处理的“中断向量”，一般把这个“中断向量”设计成一个固定的中断向量表，从 0x0 开始顺序存放，中断向量表（异常向量表）里面存放对应异常服务程序的入口地址。

回顾一下 ARM 有多少工作模式？答案是一般有 7 种。当然有的有 8 种，比如 ARMv7-A 版本就有 Secure monitor 模式。这里我们以 arm926ej-s 处理器核为例来讲，ARM 有 7 种工作模式：

USR 模式: 正常用户模式，程序正常执行模式；

FIQ 模式(Fast Interrupt Request): 处理快速中断，支持高速数据传送或通道处理；

IRQ 模式: 处理普通中断；

SVC 模式（Supervisor）：操作系统保护模式，处理软件中断 swi reset；

ABT 中止模式（Abort mode）{数据、指令}: 处理存储器故障、实现虚拟存储器和存储器保护；

UND 未定义模式（Undefined）：处理未定义的指令陷阱，支持硬件协处理器的软件仿真;

SYS 系统模式(基本上=USR)（System）：运行特权操作系统任务。

在异常服务程序里，**需要切换工作模式**，这点需要注意，否则无法操作成功。这种自动转移机制在系统内存 0x0 开始处按如下顺序设计一个中断向量表，在 PC 机里面我们把这个叫中断向量表，如果是 ARM 可以叫异常向量表。

图 1 异常向量表

一共八个异常向量，从 0x0 开始存放，每个向量是一个 32 位的地址，对应异常服务程序的入口，占 4 个字节。计算机上电后会自动进入复位异常服务程序，在复位异常程序开始，我们完成软硬件环境初始化后，一般会将系统交给高级语言编写的操作系统软件去接管计算机的运行。

2 开始裸机系统编程仿真

明白了 ARM 异常向量机制，我们开始编写一个最简单的系统程序，首先编写可直接操作硬件的 ARM 汇编程序 start.S。

```
.section INTERRUPT_VECTOR, "x"
.global _Reset
```

```
.data

.text
.code 32

_Reset:
  B Reset_Handler /* Reset */
  B . /* Undefined */
  B . /* SWI */
  B . /* Prefetch Abort */
  B . /* Data Abort */
  B . /* reserved */
  B . /* IRQ */
  B . /* FIQ */

Reset_Handler:
  LDR sp, =stack_top
  BL c_entry
  B .

.end
```

这个程序申明了一个全局入口_Reset，然后在这个下面，存放的就是异常向量表，一共 8 个 B 语句，这里只有复位异常我们写了个最简单的服务程序 Reset_Handler，其他 7 个没做处理，在 ARM 汇编中 PC 或者"."表示当前指令地址，所以"B ."就是个死循环。

Reset_Handler 进入复位后，我们简单用 LDR 指令将 sp 堆栈寄存器指向了在链接文件(map.lds)里面设置的栈顶地址，然后 BL(branch link)指令跳转到 c_entry 入口，这是我们在 init.c 文件写的一个入口函数。

init.c 程序代码如下：

```
void __aeabi_unwind_cpp_pr0 (void) {}
void __aeabi_unwind_cpp_pr1 (void) {}

volatile unsigned int * const UART0DR = (unsigned int *)0x101f1000;

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

```
int c_entry()
{
    print_uart0("Hello world!\n");
}
```

在这个 C 语言程序里，我们定义了串口 uart0 的数据寄存器的端口地址，给它的数据寄存器里面发字符，当然我们在汇编里省略了串口的初始化设置，实际是不能省略的，否则是不起作用的，但目前我们是软件仿真，它是可以出来的。

我还是不习惯一句句单独去编译、链接，我直接写一个 Makefile，当然我们同样会用 ddd/gdb 去调试运行，而不是直接运行就完。

Makefile 文件如下：

```
all:
    arm-none-linux-gnueabi-as -mcpu=arm926ej-s -g start.S -o start.o
    arm-none-linux-gnueabi-gcc -c -mcpu=arm926ej-s -g init.c -o init.o
# -nostdlib
    arm-none-linux-gnueabi-ld -T map.lds init.o start.o -o test.elf
    arm-none-linux-gnueabi-objcopy -O binary test.elf test.bin

run: all
    qemu-system-arm -M versatilepb -m 128M -gdb tcp::1024 -serial stdio
-kernel test.elf -S &
    sleep 3
    ddd --debugger arm-none-linux-gnueabi-gdb test.elf
    #qemu-system-arm -M versatilepb -serial stdio -kernel test.bin

clean:
    rm -rf *.o test.elf test.bin
```

大家可以看到这回我不是用 arm-linux-gcc 通吃编译了，而是 C 语言用 arm-none-linux-gnueabi-gcc 这个交叉编译器工具链里的 C/c++编译器，汇编则用 arm-none-linux-gnueabi-as，指定了处理器核，使用 arm-none-linux-gnueabi-ld 链接器将具有调试信息的两个目标文件 init.o 和 start.o 链接成 elf 文件，用 objcopy 生成二进制 bin 文件。ELF 格式是可执行文件、可重定位文件、共享库 object(又叫做共享库)文件。用作链接器的输入生成可执行的映像文件(bin)、可装载到内存里运行。

大家可以看到 ld 命令需要写一个链接文件，我们写的是 map.lds，写这个文件我简单讲一下一个计算机程序的组成。

计算机程序一般由 **bss 段、data 段、text** 三个段组成的，在 PC 机下我们对这个没有概念，但在嵌入式系统设计中，它涉及嵌入式系统运行时的内存大小分配，存储单元占用空间大小等问题。

在采用段式内存管理的体系结构中，bss 段通常是指用来存放程序中未初始化的全局变量的一块内存区域，一般在初始化时 bss 段部分将会清零。bss 段属于静态内存分配，即程序一开始就将其清零了。

而 C 语言之类的程序编译完成之后，已初始化的全局变量保存在.data 段中，未初始化的全局变量保存在.bss 段中。text 和 data 段分布在可执行文件中，由系统从可执行文件中读取加载，而 bss 段一般由系统进行初始化。

如果涉及 C 语言的运行环境，还涉及到堆（heap）和栈(stack)的区别。

C 语言的堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。比如进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

C 语言的栈属于先进先出(FIFO)结构，是用户存放程序临时局部变量的地方，比如函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在 data 数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。栈，所以栈特别方便用来保存/恢复调用现场。

明白这些后，下面我们写第一个链接文件 map.lds：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*指定输出可执行文件 elf 格式，32 位 ARM 指令，小端模式*/
OUTPUT_ARCH(arm)    /*指定体系结构为 ARM*/
ENTRY(_Reset)   /*指定输出可执行文件的起始入口为_Reset*/

SECTIONS
{
 . = 0x10000;   /*定位当前地址为 0x10000 地址*/
 .start . : { start.o(.text) }  /*第一个代码段来自目标文件 start.o*/
 .text : { *(.text) }   /*其他代码段*/
 .data : { *(.data) }   /*指定读写数据段*/
 .bss : { *(.bss COMMON) }  /*指定 bss 段 静态内存分配*/
 . = ALIGN(8);  /*以八字节对齐*/
 . = . + 0x1000; /* 4kB of stack memory */
```

```
 stack_top = .; /*指定栈顶指针地址*/
}
```

每句后面我都做了注释，所以我不做过多讲解了，只是在 sections 段，定位当前地址为 0x10000 地址，可能有同学会问，老师你不是说异常向量表是从 0x0 开始的么，计算机不是也是从 0x0 开始运行的么，怎么这里是 0x10000，这是因为我们是准备用 qemu 仿真运行，而 qemu 仿真运行程序的首地址就是 0x10000, 而不是 0x0，这样讲大家就明白了，后面如果我们讲用 bootloader 或者操作系统去加载单独的程序到内存运行的时候，这个地址将是期望的内存地址，当然这是后话，这里不多讲了。

至此，第一个系统仿真程序我们编完了，下面我们要开始系统仿真了。

3 ARM 系统软件仿真调试

前面的软件仿真，我们一直用的是 qemu-arm 和 arm-linux-gdb 仿真调试，没有涉及系统仿真，这次我们用 qemu-system-arm 和 arm-linux-gdb 联合仿真调试，调试语句如下(在 Makefile 文件里)：

```
run: all
    qemu-system-arm -M versatilepb -serial stdio -kernel test.elf -S -g
1024 &
    sleep 3
    ddd --debugger arm-linux-gdb test.elf
```

如果我们只是想仿真运行一下，我们可以直接运行二进制可执行文件：

qemu-system-arm -M versatilepb -serial stdio -kernel test.bin

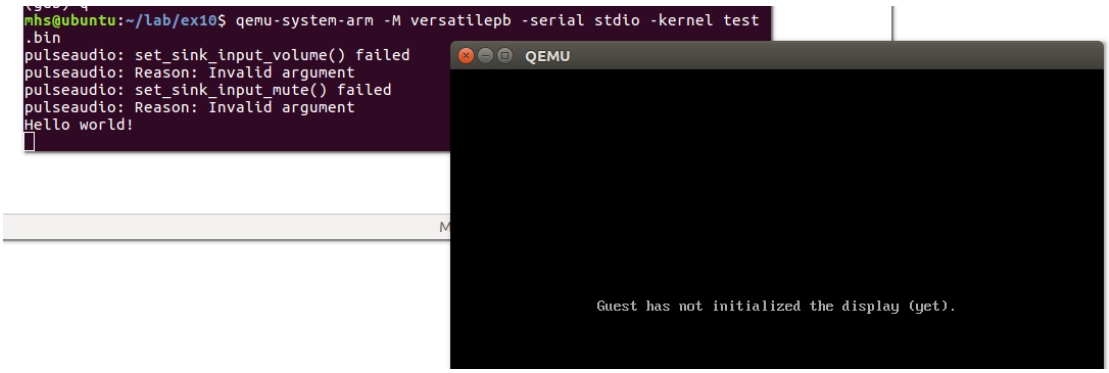

图 1 仿真运行第一个裸机系统程序

可以看到我们把 serial 定位到 stdio 设备，所以它打印出了 Hello world!

如果我们执行 make run，那么使用 ddd 会出现带 gui 的调试。

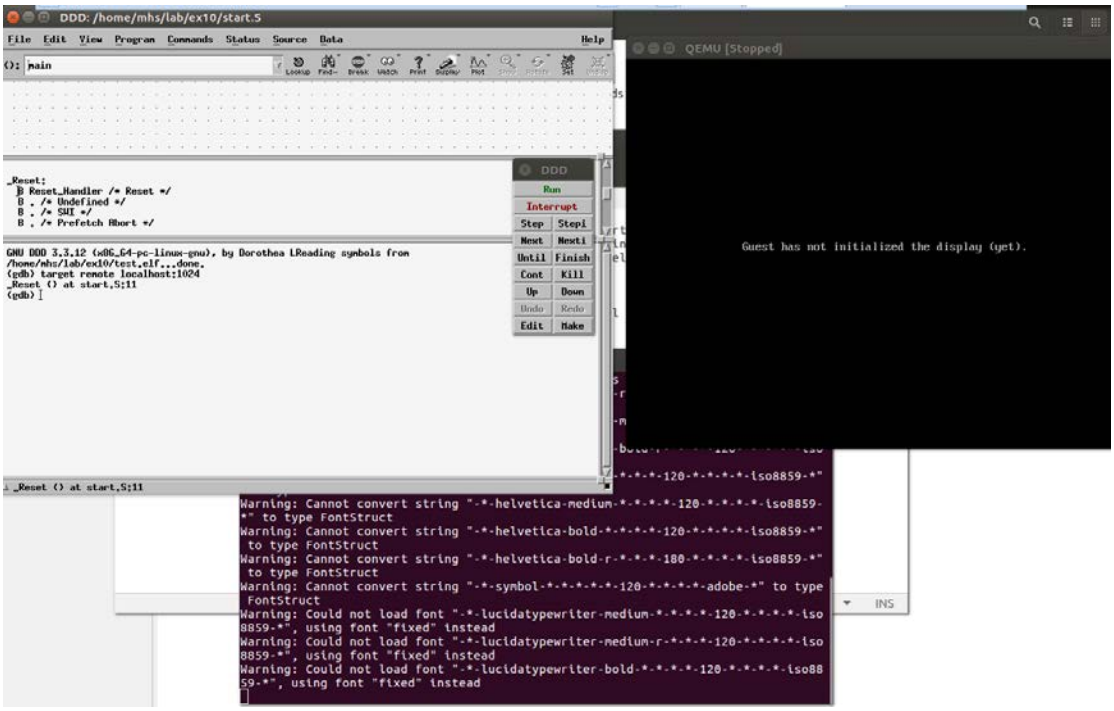

图 2　ddd 调试

大家可以看到，系统出现了一个 qemu 模拟的黑屏程序界面，我们在 ddd 里面执行 target remote localhost:1024 后，系统连上了 gdb，停留在_Reset 处，见图 3。

```
GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea LReading symbols from
/home/mhs/lab/ex10/test.elf...done.
(gdb) target remote localhost:1024
_Reset () at start.S:11
(gdb)
```

图 3 连接 target

下面我们可以用 ddd 带的调试按钮一步步去调试了，首先我们先设个断点在 c_entry，然后点击 stepi 按钮，可以看到图 4 所示画面，系统执行不了单步运行指令，出现 warning: Invalid remote reply。

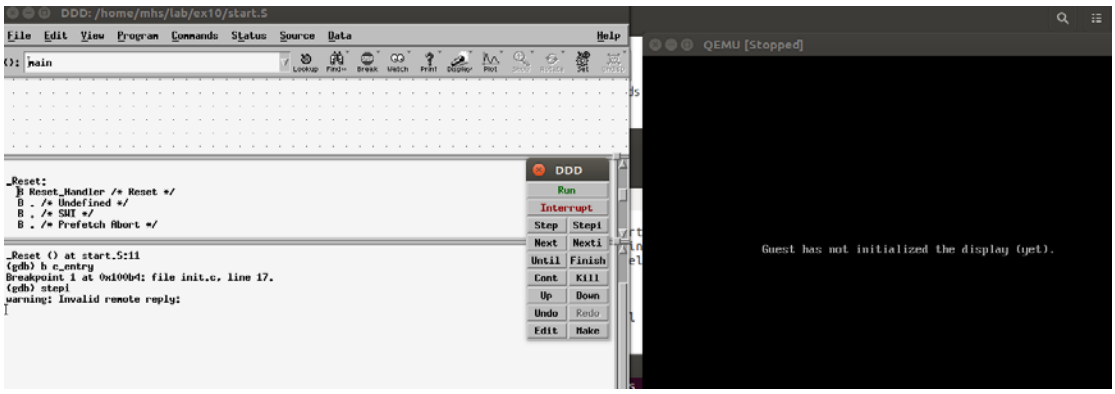

图 4 单步调试

啥意思，gdb 没有应答，这是什么原因？再看，原来那个 qemu 的 monitor 是 stopped 状态，难怪没有应答，我们定位到这个 qemu 的黑屏窗口，按 ctrl+alt+2，出现 qemu 的 monitor 可输入命令的窗口，见图 5。

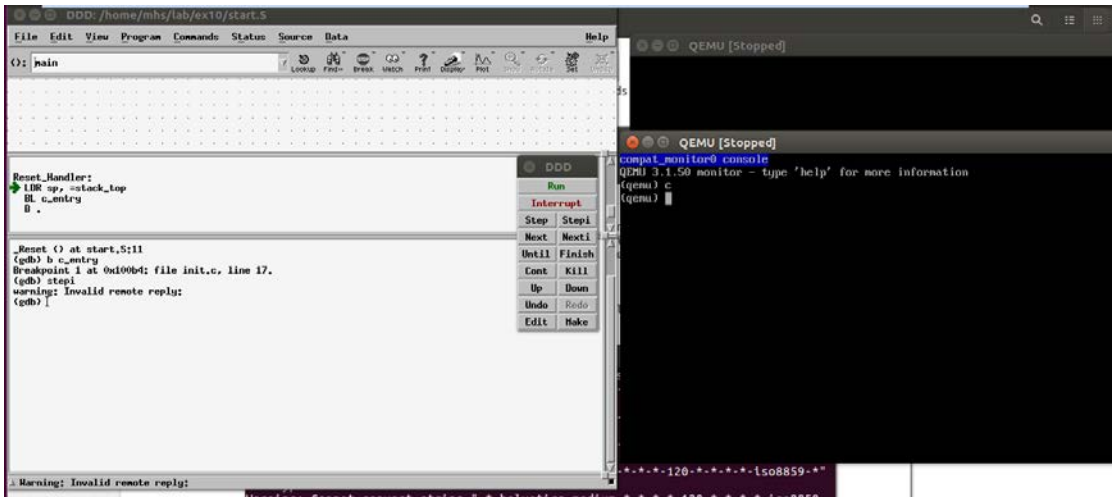

图 5 qemu monitor

可以看到如果我在 qemu 的 monitor 里敲入 c 或者 cont，它会执行反应，ddd 里面得到回答，可以单步执行了，见图 5 左边。

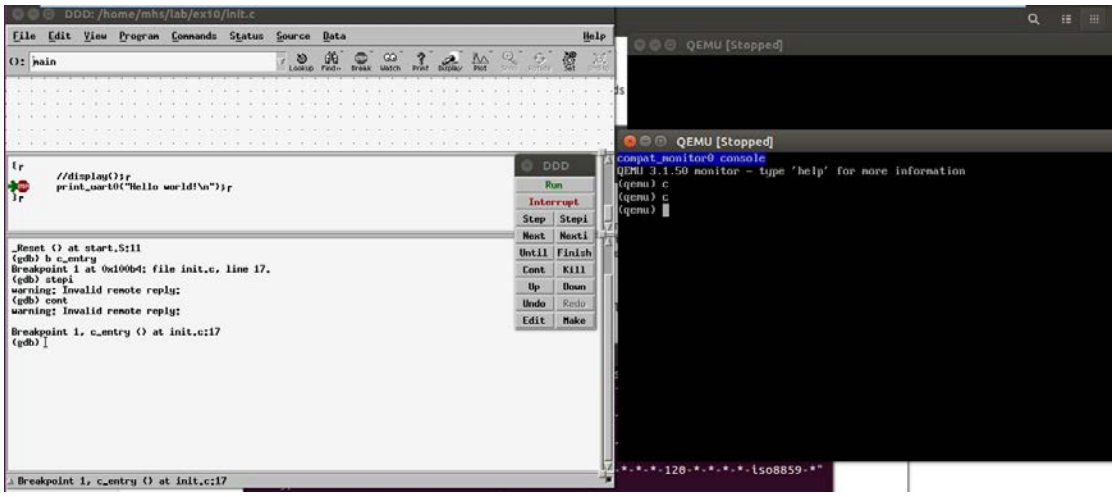

图 6 断点调试

接下来我直接在 ddd 里面点 cont，发现还是得到 monitor 里去敲 c 回车，然后我们可以看到程序连续运行，然后停留在我们设置的断点处了，怎么样，调试还是很方便的。

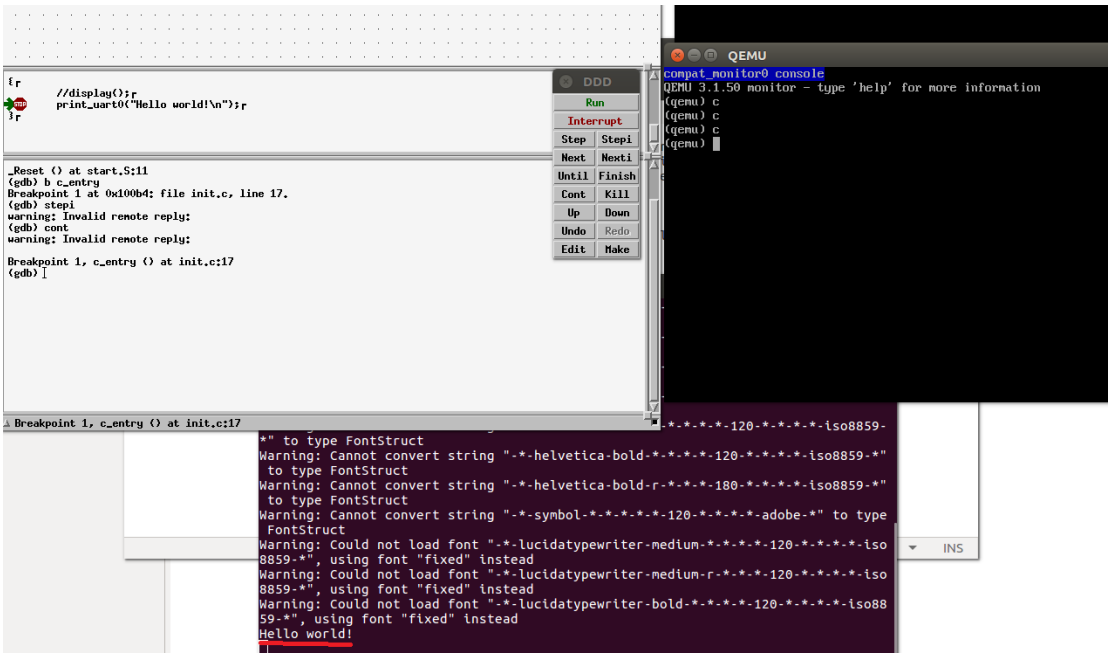

图 7 运行完打印出语句

不想一步步调了，我直接在 qemu 里敲了个 c，然后我们看到程序直接运行到底了，打印出 Hello world!，见图 7。

今天的教程就到这，我们今天进行了一个最简单的系统仿真调试，希望大家能成果掌握，多体会原理。