

树莓派挑战性任务 实验报告 (lab1-lab4)

树莓派挑战性任务 实验报告 (lab1-lab4)

LAB1 实验报告

lab1部分工作概要

lab1核心工作部分详细说明

降低Exception level至EL1与start.S的修改

用qemu和gdb实现对内核的调试

LAB2实验报告

lab2部分工作概要

lab2核心工作部分关键细节说明

建立基本的内核页表

设置并启用MMU

lab3实验报告

lab3部分工作概要

lab3核心工作部分关键细节说明

建立trapframe 和保存恢复现场方法

启用中断/异常处理程序

设置定时器启用时钟中断

实现上下文的切换

lab4实验报告

lab4部分工作概要

lab4核心工作部分关键细节说明

实现系统调用机制并移植系统调用

LAB1 实验报告

尽管下发的代码中已经包含了lab1部分，实现了printf函数，但这并不意味着我lab1就什么都没干。下面是lab1中我的工作的概要。

lab1部分工作概要

- 学习aarch64体系结构的基础知识
- 学习arm64汇编指令集的常用指令
- 安装qemu和aarch64-elf-gdb环境并学习其简单使用
- **(核心工作)** 修改start.S使Exception level从2降至1
- **(核心工作)** 修改start.S使内核在Exl1中仍可正常运行
- **(核心工作)** 修改Makefile使得编译出的内核在qemu中运行时可以用gdb远程连接进行调试
- 漫长的debug

lab1核心工作部分详细说明

降低Exception level至EL1与start.S的修改

在我们的操作系统实验中，内核应当在EL1（内核态）和EL0(用户态)中运行。然而在qemu中，最初内核是在EL2中被运行的，因此我们需要执行eret指令使其回落到el1。

然而，只执行eret指令，内核是无法进行运行的，我们需要运行一些代码对处理器进行设置，使得内核可以正常运行。例如以下关键片段

```
1      mov     x0, #0x33FF
2      msr     cptr_el2, x0
3      msr     hstr_el2, xzr
4      mov     x0, #(0xf << 20)
5      msr     cpacr_el1, x0
```

这段代码的作用是通过设置特定的寄存器，允许处理器运行SIMD指令而不触发异常，因为我们在编译内核时，交叉编译器总是会编译出此类指令作为对代码的优化。没有这段代码，内核就不能运行。

```
1      mov     x2, #0x004
2      msr     spsr_el2, x2
```

以上这段代码这则是对EL1下的PSTATE寄存器进行设置。

```
1      adr     x2, 5f
2      msr     elr_el2, x2
3      eret
```

以上这段代码则是使得回落到EL1时，可以跳转的正确的地址运行。

为了检验我们确实把内核降至el1，我们通过以下函数打印当前的exception level

```
1  .global get_el
2  get_el:
3      mrs     x0,CurrentEl
4      lsr     x0,x0,#2
5      ret
```

```
1  #include <printf.h>
2  extern int get_el();
3  void printel(){
4      printf("Current exception level switched to: %d \r\n",get_el());
5  }
```

我们可以看到以下内容输出

```
qemu-system-aarch64 -M raspi3 -serial stdio -kernel kernel8.img
VNC server running on 127.0.0.1:5900
>>>main.c:      main is start ...>>>
Current exception level switched to: 1
```

用qemu和gdb实现对内核的调试

aarch64的工具链中提供了aarch64-el-gdb工具。我们可以用这个工具连接在debug状态下的qemu模拟器实现对内核的单步调试。

首先，我们对include.mk进行修改，在kernel.elf中保留debug信息，供gdb使用。

需要debug时我们运行以下指令

```
1 | qemu-system-aarch64 -S -s -M raspi3 -serial stdio -kernel kernel8.img
```

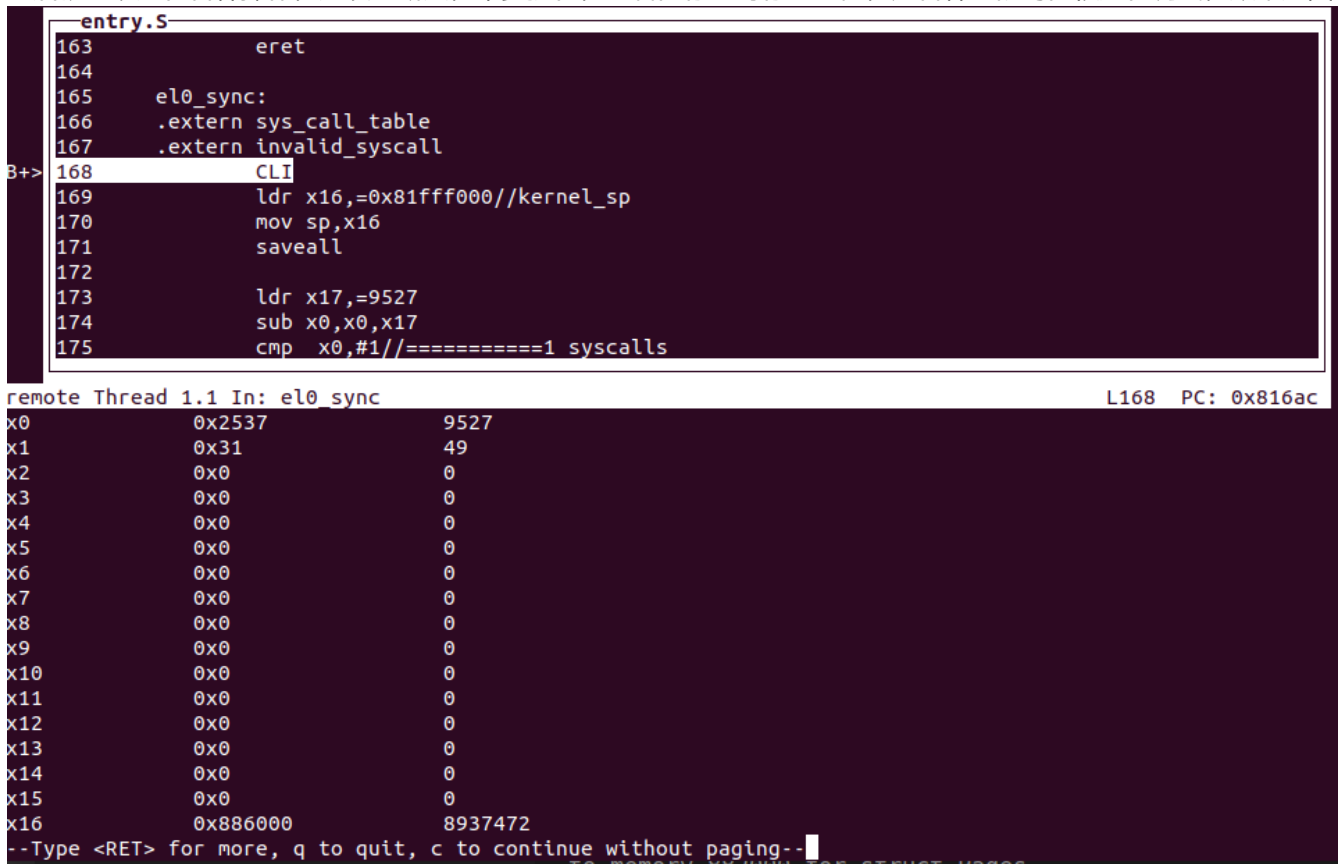
然后新开一个terminal，运行以下命令打开gdb。

```
1 | aarch64-elf-gdb kernel.elf
```

最后，在gdb连接qemu模拟器。

```
1 | target remote localhost:1234
```

之后就可以进行各种操作，如设置断点，单步执行，查看内存与寄存器的值，用各种手段对内核进行调试，效果如图



The screenshot shows a GDB session with the following assembly code loaded in the entry.S file:

```
163          eret
164
165     el0_sync:
166     .extern sys_call_table
167     .extern invalid_syscall
168     CLI
169     ldr x16,=0x81fff000//kernel_sp
170     mov sp,x16
171     saveall
172
173     ldr x17,=9527
174     sub x0,x0,x17
175     cmp x0,#1//=====1 syscalls
```

The GDB prompt shows the current thread is 1.1, in the el0_sync function, at address L168, PC: 0x816ac. The register values are displayed as follows:

Register	Value	Comment
x0	0x2537	9527
x1	0x31	49
x2	0x0	0
x3	0x0	0
x4	0x0	0
x5	0x0	0
x6	0x0	0
x7	0x0	0
x8	0x0	0
x9	0x0	0
x10	0x0	0
x11	0x0	0
x12	0x0	0
x13	0x0	0
x14	0x0	0
x15	0x0	0
x16	0x886000	8937472

The GDB prompt also shows the command: --Type <RET> for more, q to quit, c to continue without paging--

LAB2实验报告

在这一部分主要进行的是mmu的设置与启动，以及页式管理系统的设置。经过这一部分的实验后，我们的操作系统具备了对虚拟内存的使用与管理能力。

lab2部分工作概要

- 设计在本次树莓派任务中所使用的内存分配方案
- （核心工作）建立基本的内核页表

- **(核心工作)** 对控制mmu的相关寄存器进行设置，正确启用mmu
- **(核心工作)** 修改并移植MIPS操作系统中，lab2的有关内容
- 编写page_check()函数检查正确性
- 漫长的debug

lab2核心工作部分关键细节说明

建立基本的内核页表

由于树莓派中并没有类似于mips操作系统实验中，keseg0这种，在启用mmu的情况下直接访存的机制，这也就意味着，我们必须在启用mmu之前，预先配置好可以供mmu使用的，内核地址的各级页表。为此，我编写了一个boot_mmu_setup()函数，用来实现这一功能。在该操作系统中我们采取的是三级页表设置，分别叫做PUD(page upper directory),PMD(page middle directory),PGT(page table)，每个页面大小为4k，每个页表页有512个页表项，一个PUD对应512G，一个PMD对应1G，一个PGT对应2M 0x3f000000-0x40000000+4k的区域是设备内存，其他内存均为常规内存。

设置并启用MMU

以下为实现这一步骤的代码

```

1  .global enable_mmu
2  enable_mmu:
3      adrp    x0, _pg_dir
4      msr ttbr0_el1, x0
5
6      ldr x0, =0x440488
7      msr mair_el1,x0
8      //msr mair_el0,x0
9
10     ldr x0, =(TCR_VALUE)
11     msr tcr_el1, x0
12
13     mrs x0,sctlr_el1
14     orr x0,x0,#0x1
15     msr sctlr_el1,x0
16
17     ret

```

这个汇编函数中，第一小节是在装载最高级页目录，将其物理地址存入到ttbr0_el1这一寄存器中。

第二小节中，是设置了mair寄存器，这一寄存器在指导书中进行了介绍，是设置不同种类内存的的寄存器。

第三小节中，对tcr_el1寄存器的设置，可以用来对el1高位与低位地址的属性（如cache类型，虚拟内存结构，可见性等.....）

第四小节，启用mmu.

lab3实验报告

在lab3中我们引入了时钟中断和进程，使得OS可以同时运行多个进程。

lab3部分工作概要

- **(核心工作)** 重新建立使用于树莓派的trapframe,与保存/恢复现场方法

- **(核心工作)** 修改并移植关于进程控制的代码
- 编写env_check()检查正确性
- **(核心工作)** 启用中断与异常的处理程序
- **(核心工作)** 设置定时器并打开时钟中断
- **(核心工作)** 实现上下文的切换
- 进行测试，实现和MIPS一样的，经典的输出1和2现象
- 漫长的debug
- 有正确现象了，跟女朋友大吃一顿香锅庆祝一下

lab3核心工作部分关键细节说明

建立trapframe 和保存恢复现场方法

本次使用的trapeframe如下：

```
1 struct Trapframe{
2     unsigned long x[31]; //register x0-30
3     unsigned long sp;
4     unsigned long elr_el1; //elr_el1 register, in mips we called epc
5     unsigned long pstate;
6     unsigned long pc;
7 };
```

x[31]是为了保存X0-X30这三十个寄存器，sp用来保存el0状态下的栈指针，elr_el1寄存器相当于mips中的epc寄存器，pstate保存有关于条件跳转，时钟开关，异常级等关键信息，必须保存，恢复现场时，会使用pc这个值。我们定义了两个汇编宏用于保存/恢复现场，内容如下

```
1 .macro saveall
2     sub sp, sp, #280
3     stp x0, x1, [sp, #16 * 0]
4     stp x2, x3, [sp, #16 * 1]
5     stp x4, x5, [sp, #16 * 2]
6     stp x6, x7, [sp, #16 * 3]
7     stp x8, x9, [sp, #16 * 4]
8     stp x10, x11, [sp, #16 * 5]
9     stp x12, x13, [sp, #16 * 6]
10    stp x14, x15, [sp, #16 * 7]
11    stp x16, x17, [sp, #16 * 8]
12    stp x18, x19, [sp, #16 * 9]
13    stp x20, x21, [sp, #16 * 10]
14    stp x22, x23, [sp, #16 * 11]
15    stp x24, x25, [sp, #16 * 12]
16    stp x26, x27, [sp, #16 * 13]
17    stp x28, x29, [sp, #16 * 14]
18    str x30, [sp, #16 * 15]
19    mrs x16, sp_el0
20    str x16, [sp, #8*31]
21    mrs x16, elr_el1
22    str x16, [sp, #8*32]
23    str x16, [sp, #8*34] //pc
24
```

```

25     mrs x16,spsr_el1
26     str x16,[sp,#8*33]
27 .endm
28
29 .macro restore
30     ldp x0, x1, [sp, #16 * 0]
31     ldp x2, x3, [sp, #16 * 1]
32     ldp x4, x5, [sp, #16 * 2]
33     ldp x6, x7, [sp, #16 * 3]
34     ldp x8, x9, [sp, #16 * 4]
35     ldp x10, x11, [sp, #16 * 5]
36     ldp x12, x13, [sp, #16 * 6]
37     ldp x14, x15, [sp, #16 * 7]
38     ldp x16, x17, [sp, #16 * 8]
39     ldp x18, x19, [sp, #16 * 9]
40     ldp x20, x21, [sp, #16 * 10]
41     ldp x22, x23, [sp, #16 * 11]
42     ldp x24, x25, [sp, #16 * 12]
43     ldp x26, x27, [sp, #16 * 13]
44     ldp x28, x29, [sp, #16 * 14]
45     ldr x30, [sp, #16 * 15]
46
47     ldr x16,[sp,#8*31]
48     msr sp_el0,x16
49
50     ldr x16,[sp,#8*33]
51     msr spsr_el1,x16
52
53     ldr x16,[sp,#8*34]//pc
54     msr elr_el1,x16
55     add sp, sp, #280
56     //eret
57 .endm

```

启用中断/异常处理程序

首先，建立异常/中断处理向量，在每个向量中保存可以调转至对应处理程序的汇编指令,如下 后面摆放各种的异常/中断处理程序的调转指令就好啦.

```

1  .align 11
2  .global vectors
3
4  vectors:
5      handler_sync_invalid_el1t

```

启用中断/异常处理程序本质就是启用这张异常向量表，通过如下代码实现，就是把vectors地址装进了vbar_el1这个寄存器中去。

```

1  .globl irq_vector_init
2  .extern vectors
3  irq_vector_init:
4      adr x0, vectors          // load VBAR_EL1 with virtual
5      msr vbar_el1, x0        // vector table address
6      ret

```

设置定时器启用时钟中断

首先，按照我们指导书的方法，我们设置了定时器

```

1  .global timer_init
2  timer_init:
3      mov x0, #0x3
4      msr cntkctl_el1, x0
5      ldr x0, =(0x3b9aca0>>6)
6      msr cntp_tval_el0, x0
7      mov x0, #0x1
8      msr cntp_ctl_el0, x0
9      ret

```

cntkctl_el1设置el0和el1皆可访问相关寄存器。

时钟频率就是0x3b9aca0好像改不了，但是可通过设置cntp_tval_el0寄存器的值来控制几个时钟周期触发一次中断，达到变相调节频率的作用。

设置cntp_ctl_el0来启用时钟。

再调用void enable_interrupt_controller()这个C函数，向0x40000040 0x40000044 0x40000048 0x4000004c这四个地址写入0xf1以打开对所有时钟中断的响应。值得一提的是，树莓派这个时钟中断是个一锤子买卖，所以每次时钟中断要重置时钟。然后编写相应的中断处理函数，写好调度函数即可。

实现上下文的切换

env_pop_tf函数实现上下文的切换，这肯定是不必说的。

烦人的是这个cache，每次切换上下文之前必须更新cache

我比较暴力，每次切换上下文就让所有cache作废

代码如下

```

1  .globl tlb_invalidate
2  tlb_invalidate:
3      dsb ishst                // ensure write has completed
4      tlbi vmlalleis          // invalidate tlb, all asid, el1.
5      dsb ish                  // ensure completion of tlb invalidation
6      isb                      // synchronize context and ensure that no instructions
7                               // are fetched using the old translation
8      ret

```

lab4实验报告

lab4中主要引入了系统调用机制，并引入了一个简单的fork。

说明：受限于时间不足，在这里的fork我并没有采用写时复制的机制，也没有遵守微内核的精神。这是我的移植工作中的一大不足

lab4部分工作概要

- (核心工作) 实现系统调用机制。
- (核心工作) 移植MIPS操作系统上的系统调用
- 测试各个系统调用是否工作正常
- 实现一个假的fork

lab4核心工作部分关键细节说明

实现系统调用机制并移植系统调用

除了Mips中的syscall指令在这里被换成了svc #0指令外，其他的内容大体上都与Mips的系统调用内容相同，如下所示。

```
1  .global msyscall
2  msyscall:
3      svc #0
4      ret
```

分派系统调用的方法也与Mips大同小异

```
1  e10_sync:
2  .extern sys_call_table
3  .extern invalid_syscall
4      CLI
5      ldr x16,=0x51fff000//kernel_sp
6      mov sp,x16
7      saveall
8
9      ldr x17,=9527
10     sub x0,x0,x17
11     cmp x0,#20//=====20 syscalls
12     b.ge invalid_syscall
13
14     adr x17,sys_call_table
15     add x17,x17,x0,lsl #3
16     ldr x17,[x17]
17     blr x17
18
19     restore
```