南京大学
NANJING UNIVERSITY

RT-Thread移植

陈璐

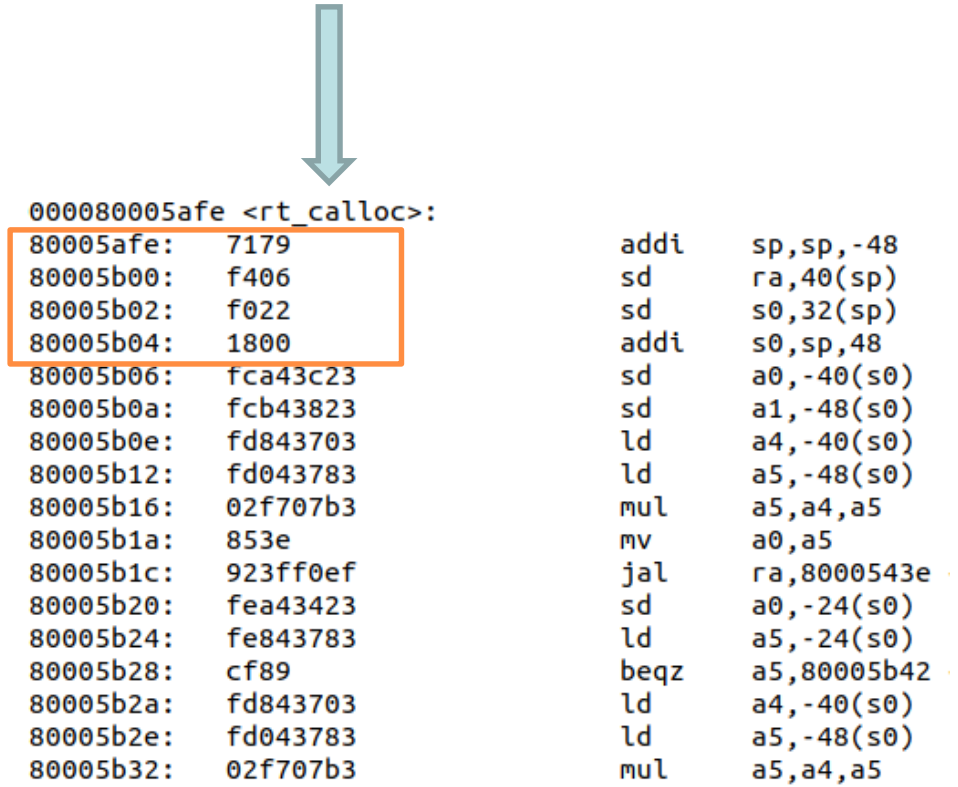
2021.8.18

提纲

- 移植内容&使用方法
- RT-Thread文件组织
- 移植到GNU工具链
- 将中断改为轮询

运行官方RT-Thread

- 官方RT-Thread编译（**不建议**）
  - 获得riscv Newlib交叉编译工具的两种方法
    - 直接下载SiFive提供的rv64gc工具链：指令集与一生一芯不兼容
    - 自行编译：繁琐且困难
  - 在rtconfig.py中设置安装路径
  - 使用scons编译



RT-Thread移植目标

- 移植到GNU工具链：可一键安装工具链，无需自行编译
- 去除压缩指令和乘除指令：符合一生一芯要求指令集
- 通过轮询访问uart：避免引入外部中断，简化实现

- 我们已经帮助大家完成移植工作
  - 移植后的RT-Thread编译过程

```
1  sudo apt-get install scons g++-riscv64-linux-gnu binutils-riscv64-linux-gnu
2  git clone https://github.com/OSCPU/rt-thread.git
3  cd rt-thread/bsp/qemu-riscv-virt64
4  scons
```

提纲

- 移植内容&使用方法
- RT-Thread文件组织
- 移植到GNU工具链
- 将中断改为轮询

RT-Thread文件组织

```
rt-thread
├── bsp                    //板级支持包
│   ├── nucleiS
│   ├── qemu-riscv-virt64
│   ├── qemu-vexpress-a9
│    ......
├── components             //组件
├── examples
├── include
├── libcpu                 //芯片移植相关文件
│   ├── arm
│   ├── risc-v
│   ├── xilinx
│   ......
└── src                    //内核层
    ├── clock.c
    ├── cpu.c
    ├── device.c
    ├── irq.c
    ├── kservice.c
    ├── mem.c
    ├── scheduler.c
    ├── signal.c
    ├── thread.c
    ......
```

- 板级支持包：对硬件进行抽象，主要包含外设驱动，如uart
- 组件：基于RT-Thread内核的上层软件，如FinSH命令行界面、虚拟文件系统
- 芯片移植相关文件：可类比于AM中的TRM+IOE+CTE
- RT-Thread内核：包括线程调度、内存管理等

提纲

- 移植内容&使用方法
- RT-Thread文件组织
- 移植到GNU工具链
- 将中断改为轮询

更换工具链

- 在rtconfig.py设置交叉编译工具链路径

```
18      -   EXEC_PATH   = r'/home/lizhirui/workspace/riscv64-toolchains/bin'
    18  +   EXEC_PATH   = r'/usr/bin'
30      -   PREFIX  = 'riscv64-unknown-elf-'
    30  +   PREFIX  = 'riscv64-linux-gnu-'
```

  - apt-get默认将gcc等可执行文件安装在/usr/bin下

- 编译错误

```
/usr/riscv64-linux-gnu/include/bits/types/__sigval_t.h:24:7: error: redefinition of 'union sigval'
   24 | union sigval
      |       ^~~~~~
In file included from /home/piper/program/rt-thread/include/rtlibc.h:19,
                 from /home/piper/program/rt-thread/include/rtdef.h:1154,
                 from /home/piper/program/rt-thread/include/rtthread.h:25,
                 from applications/main.c:11:
/home/piper/program/rt-thread/include/libc/libc_signal.h:28:7: note: originally defined here
   28 | union sigval
      |       ^~~~~~
In file included from /usr/riscv64-linux-gnu/include/signal.h:57,
                 from /home/piper/program/rt-thread/include/libc/libc_signal.h:72,
                 from /home/piper/program/rt-thread/include/rtlibc.h:19,
                 from /home/piper/program/rt-thread/include/rtdef.h:1154,
                 from /home/piper/program/rt-thread/include/rtthread.h:25,
                 from applications/main.c:11:
```

解决头文件相关问题

- 部分库没有提供完整的信号机制，rt-thread在编译时会首先去/usr/lib下查找是否存在信号相关的头文件，不存在就会使用rt-thread内部定义的结构信息
  - 但头文件实际存在于/usr下
  - 修改路径、解决头文件问题
- 编译成功了

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
LINK rtthread.elf
riscv64-linux-gnu-objcopy -O binary rtthread.elf rtthread.bin
riscv64-linux-gnu-size rtthread.elf
   text    data     bss     dec     hex filename
 120231    1816   70412  192459   2efcb rtthread.elf
scons: done building targets.
```

```
def GetGCCRoot(rtconfig):
    exec_path = rtconfig.EXEC_PATH
    prefix = rtconfig.PREFIX

    if prefix.endswith('-'):
        prefix = prefix[:-1]

    if exec_path == '/usr/bin':
        root_path = os.path.join('/usr/lib', prefix)
    else:
        root_path = os.path.join(exec_path, '..', prefix)

    return root_path
```

运行试试

- 使用目录下的脚本，在qemu中运行rt-thread

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ sh qemu-nographic.sh
```

  - 没有任何输出?

```
la   sp, __stack_start__
li   t0, __STACKSIZE__
```

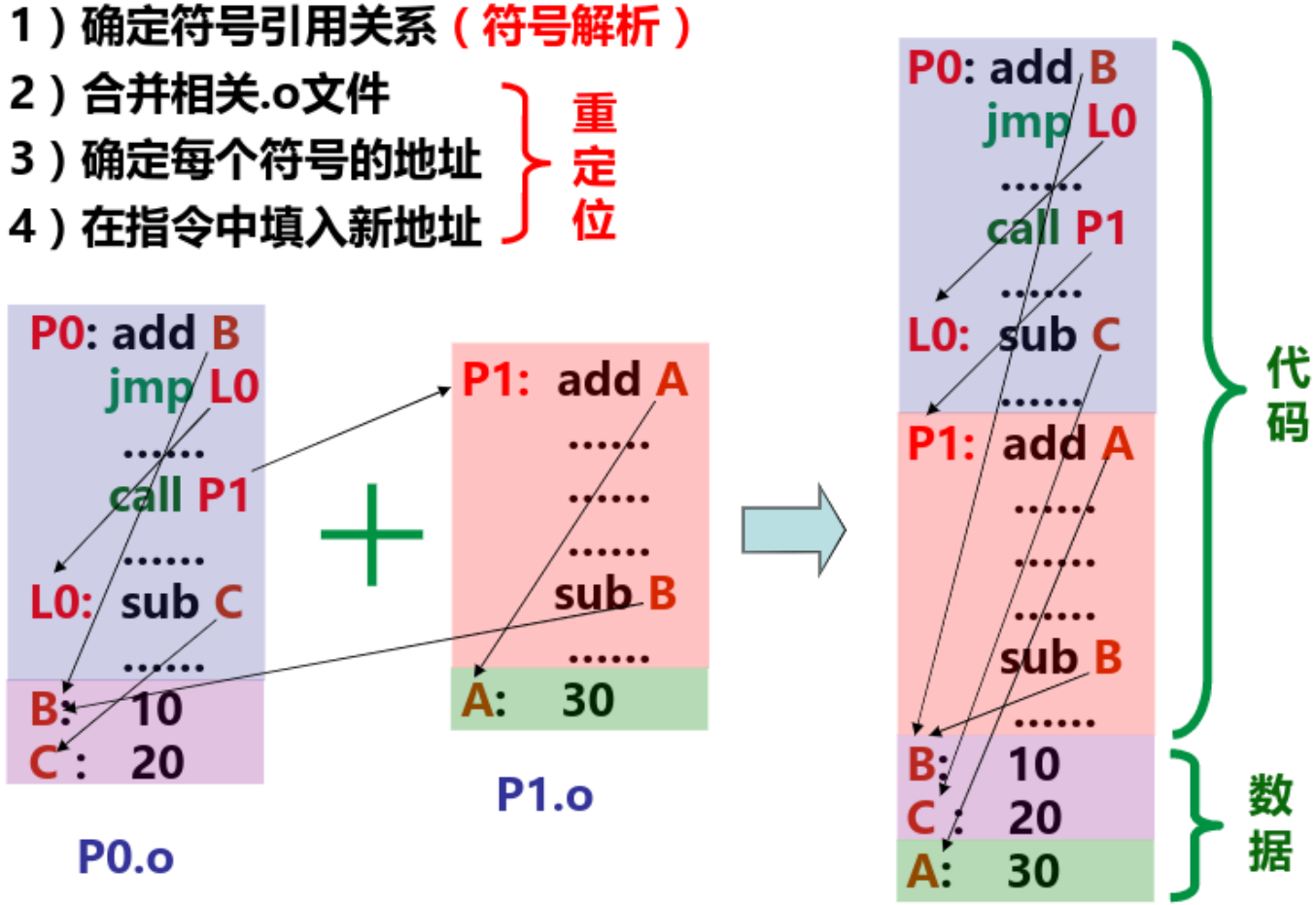
```
8000002e:   00019117            auipc   sp,0x19
80000032:   b7213103            ld      sp,-1166(sp) # 80018ba0 <_GLOBAL_OFFSET_TABLE_+0x38>
```

  - 在_start中会初始化栈指针sp，初始值居然是0
    - 需要重定位的表项
    - 后续压栈过程出错
  - 理解这个错误需要对链接有基本的了解

```
0000000080018b68 <.got>:
    80018b68:   89c8                    0x89c8
    80018b6a:   8001                    c.srli64        s0
        ...
                80018b70: R_RISCV_RELATIVE      *ABS*+0x80029d98
                80018b78: R_RISCV_RELATIVE      *ABS*+0x80020d88
                80018b80: R_RISCV_RELATIVE      *ABS*+0x80029da8
                80018b88: R_RISCV_RELATIVE      *ABS*+0x800210f0
                80018b90: R_RISCV_RELATIVE      *ABS*+0x80018450
                80018b98: R_RISCV_RELATIVE      *ABS*+0x800210e8
                80018ba0: R_RISCV_RELATIVE      *ABS*+0x80018cc0
                80018ba8: R_RISCV_RELATIVE      *ABS*+0x80018450
                80018bb0: R_RISCV_RELATIVE      *ABS*+0x80009580
```

链接器的工作(1)

- 将多个可重定位文件(.o)和静态库文件(.a)合并为可执行文件
- 符号解析
  - 将符号的定义和引用关联起来
- 重定位
  - 合并相同的节
  - 对符号定义重定位
  - 对引用符号进行重定位



袁春风，南京大学《计算机系统基础》第四章 程序的链接

链接器的工作(2)

- 符号解析：将符号的定义和引用关联起来
  - .o文件的符号可以通过nm查看
- 重定位：为每个符号分配一个内存地址
  - 合并相同的节
  - 对符号定义和引用符号进行重定位
  - 对引用符号进行重定位

```
                 U primary_cpu_entry
                 U __stack_start__
0000000000000000 T _start
                 U trap_entry
```

静态链接：程序构建时进行上述工作
动态链接：程序运行时进行上述工作

```
0000000000000000 <_start>:
   0:   f14022f3                csrr    t0,mhartid
   4:   00a29293                slli    t0,t0,0xa
   8:   f1402573                csrr    a0,mhartid
   c:   04051063                bnez    a0,4c <park>
                        c: R_RISCV_BRANCH       park
  10:   30405073                csrwi   mie,0
  14:   34405073                csrwi   mip,0
  18:   00000297                auipc   t0,0x0
                        18: R_RISCV_PCREL_HI20  trap_entry
                        18: R_RISCV_RELAX       *ABS*
  1c:   00028293                mv      t0,t0
                        1c: R_RISCV_PCREL_LO12_I        .L0
                        1c: R_RISCV_RELAX       *ABS*
  20:   30529073                csrw    mtvec,t0
```

```
0000000080000000 <_start>:
   80000000:    f14022f3    csrr    t0,mhartid
   80000004:    00a29293    slli    t0,t0,0xa
   80000008:    f1402573    csrr    a0,mhartid
   8000000c:    02051e63    bnez    a0,80000048 <park>
   80000010:    30405073    csrwi   mie,0
   80000014:    34405073    csrwi   mip,0
   80000018:    00009297    auipc   t0,0x9
   8000001c:    2f428293    addi    t0,t0,756 # 8000930c <trap_entry>
   80000020:    30529073    csrw    mtvec,t0
```

回到刚才的问题 - 静态链接和动态链接

```
la   sp, __stack_start__
li   t0, __STACKSIZE__
```

```
8000002e:   00019117            auipc   sp,0x19
80000032:   b7213103            ld      sp,-1166(sp) # 80018ba0 <_GLOBAL_OFFSET_TABLE_+0x38>
```

- 访问GOT表获取符号地址
  - GOT表可以看作一个指针数组，用于存放符号的地址，给位置无关代码进行重定位
  - 静态链接在链接阶段填入
  - 动态链接在运行时填入
- riscv64-linux-gnu-gcc默认采用动态链接
  - 动态库在程序编译时并不会被链接到目标代码中
  - 在动态链接程序运行前,系统会首先使用加载器将动态链接器加载到内存,再由动态链接器将共享库加载到内存
    - 我们编写的cpu上显然都没有

换成静态链接吧

- 加入-static链接选项
- 又失败了

```
/usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/lib/libc.a(libc_fatal.o): in function `.L0 ':
(.text+0x1a): relocation truncated to fit: R_RISCV_HI20 against symbol `__stack_chk_guard' defined in .data.rel.ro section in /usr/l
ib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/lib/libc.a(libc-start.o)
```

  - 重定位错误
- __stack_chk_guard的地址?

```
libc_fatal.o:     file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <__libc_message>:
   0:   7135                    addi    sp,sp,-160
   2:   f0a2                    sd      s0,96(sp)
   4:   f45e                    sd      s7,40(sp)
   6:   1880                    addi    s0,sp,112
   8:   f062                    sd      s8,32(sp)
   a:   f486                    sd      ra,104(sp)
   c:   eca6                    sd      s1,88(sp)
   e:   e8ca                    sd      s2,80(sp)
  10:   e4ce                    sd      s3,72(sp)
  12:   e0d2                    sd      s4,64(sp)
  14:   fc56                    sd      s5,56(sp)
  16:   f85a                    sd      s6,48(sp)
  18:   ec66                    sd      s9,24(sp)
  1a:   00000c37                lui     s8,0x0
                        1a: R_RISCV_HI20        __stack_chk_guard
                        1a: R_RISCV_RELAX       *ABS*
```

重定位地址

- 加入--print-map链接选项，输出链接映射信息

```
.data.rel.ro    0x0000000080076e48      0x10 /usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/lib/libc.a(libc-start.o)
                0x0000000080076e48                __stack_chk_guard
```

  - 符号被链接到0x80000000以上的地址空间

```
18:   ec66                    sd      s9,24(sp)
1a:   00000c37                lui     s8,0x0
                      1a: R_RISCV_HI20        __stack_chk_guard
                      1a: R_RISCV_RELAX       *ABS*
```

- lui的范围？
  - 通过apt-get下载得到的glibc默认通过-mcmodel=medlow编译
  - 通过lui获取符号地址，最多访问-2GB~2GB的地址空间

imm[31:12]	rd	0110111	LUI	x[rd] = sext(immediate[31:12] << 12)

    - $[0, 0x7fffffff] \cup [0xffffffff80000000, 0xffffffffffffffff]$

为什么是这个地址？ - 链接脚本

- 链接脚本
  - 链接脚本控制目标文件和库文件在程序地址空间内的布局
  - 变量定义、内存布局、段分布
- 查看rt-thread链接脚本link.lds
  - 链接脚本指定了起始地址为0x80000000

```
__START_ADDR__ = 0x80000000;
MEMORY
{
    SRAM : ORIGIN = __START_ADDR__, LENGTH = 0x7FF000
}

ENTRY(_start)
SECTIONS
{
    . = __START_ADDR__ ;

    /* __STACKSIZE__ = 4096; */

    .start :
    {
        *(.start);
    } > SRAM
```

验证一下(1): 使用默认链接脚本

- 去掉-T编译选项

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
LINK rtthread.elf
/usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/bin/ld: build/driver/board.o: in function `primary_cpu_entry':
/home/piper/program/rt-thread/bsp/qemu-riscv-virt64/driver/board.c:33: undefined reference to `__bss_end'
/usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/bin/ld: build/driver/board.o: in function `rt_hw_board_init':
/home/piper/program/rt-thread/bsp/qemu-riscv-virt64/driver/board.c:45: undefined reference to `__bss_end'
/usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/bin/ld: /home/piper/program/rt-thread/bsp/qemu-riscv-virt64/dr
iver/board.c:49: undefined reference to `__bss_end'
/home/piper/program/rt-thread/libcpu/risc-v/virt64/interrupt_gcc.S:108: undefined reference to `__stack_start__'
/usr/lib/gcc-cross/riscv64-linux-gnu/10/../../../../riscv64-linux-gnu/bin/ld: /home/piper/program/rt-thread/libcpu/risc-v/virt64/sta
rtup_gcc.o: in function `.L0 ':
```

  - 在链接脚本中定义

- 随便设置一下未定义的变量，看看会发生什么

```
--defsym=__bss_end=0x2000000,--defsym=__stack_start__=0x40000000
```

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build/driver/board.o
CC /home/piper/program/rt-thread/libcpu/risc-v/virt64/tick.o
LINK rtthread.elf
riscv64-linux-gnu-objcopy -O binary rtthread.elf rtthread.bin
riscv64-linux-gnu-size rtthread.elf
   text    data     bss     dec     hex filename
 422527    9910   40844  473281   738c1 rtthread.elf
scons: done building targets.
```

验证一下(2)：修改链接脚本的起始地址

- 设置起始地址为0x40000000
  - 编译也成功了
- 静态链接的GOT

```
MEMORY
{
   SRAM : ORIGIN = 0x40000000, LENGTH = 0x7FF000
}

ENTRY(_start)
SECTIONS
{
    . = 0x40000000 ;

    /* __STACKSIZE__ = 4096; */

    .start :
    {
        *(.start);
    } > SRAM
```

```
csrw SRC_XIE, 0
csrw SRC_XIP, 0

la t0, trap_entry
```

rt-thread源码

```
4000000c:    30405073                csrwi   mie,0
40000010:    34405073                csrwi   mip,0
40000014:    00072297                auipc   t0,0x72
40000018:    2542b283                ld      t0,596(t0) # 40072268 <_GLOBAL_OFFSET_TABLE_+0xd8>
```

汇编指令

```
40072268:    0cc0
4007226a:    4004
```

GOT表项

```
0000040040cc0 <trap_entry>:
 40040cc0:    7111                    addi    sp,sp,-256
 40040cc2:    e406                    sd      ra,8(sp)
```

目标地址

为什么静态链接也有GOT表

- 当编译位置无关代码时，对地址的加载过程会被扩展为对全局偏移表（GOT）的加载

- 伪指令la等同于执行
  - auipc rd， offsetHi
  - ld rd, offsetLo(rd)

```
auipc   t0,0x72
ld      t0,596(t0) # 40072268 <_GLOBAL_OFFSET_TABLE_+0xd8>
```

- 去掉GOT表
  - 加上-fno-pic编译选项禁止位置无关代码的生成

既然编译成功了...

- 在qemu上跑跑看

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ sh qemu-nographic.sh
```

  - 还是跑不起来
  - 在qemu上运行的物理内存区间为[0x80000000, 0x8fffffff]

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ sh qemu-nographic.sh
QEMU 5.2.0 monitor - type 'help' for more information
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000001000-000000000000ffff (prio 0, rom): riscv_virt_board.mrom
    0000000000100000-0000000000100fff (prio 0, i/o): riscv.sifive.test
    0000000000101000-0000000000101023 (prio 0, i/o): goldfish_rtc
    0000000002000000-000000000200ffff (prio 0, i/o): riscv.sifive.clint
    0000000003000000-000000000300ffff (prio 0, i/o): gpex_ioport
    000000000c000000-000000000c20ffff (prio 0, i/o): riscv.sifive.plic
    0000000010000000-0000000010000007 (prio 0, i/o): serial
    0000000010001000-00000000100011ff (prio 0, i/o): virtio-mmio
    0000000010002000-00000000100021ff (prio 0, i/o): virtio-mmio
    0000000010003000-00000000100031ff (prio 0, i/o): virtio-mmio
    0000000010004000-00000000100041ff (prio 0, i/o): virtio-mmio
    0000000010005000-00000000100051ff (prio 0, i/o): virtio-mmio
    0000000010006000-00000000100061ff (prio 0, i/o): virtio-mmio
    0000000010007000-00000000100071ff (prio 0, i/o): virtio-mmio
    0000000010008000-00000000100081ff (prio 0, i/o): virtio-mmio
    0000000020000000-0000000021ffffff (prio 0, romd): virt.flash0
    0000000022000000-0000000023ffffff (prio 0, romd): virt.flash1
    0000000030000000-0000000003ffffff (prio 0, i/o): alias pcie-ecam @pcie-mmcfg-mmio 0000000000000000-000000000fffffff
    0000000040000000-0000000007ffffff (prio 0, i/o): alias pcie-mmio @gpex_mmio 0000000040000000-000000007fffffff
    0000000080000000-000000008fffffff (prio 0, ram): riscv_virt_board.ram
```

```
0000040000000 <_start>:
40000000:    f14022f3                csrr    t0,mhartid
40000004:    02aa                    slli    t0,t0,0xa
40000006:    f1402573                csrr    a0,mhartid
4000000a:    ed05                    bnez    a0,40000042 <park>
```

还差些什么?

- 正确设置__bss_end和__stack_start__，以0x80000000为起始地址
  - ->使用rt-thread链接脚本
  - 但apt-get安装的glibc并不能链接到0x80000000

- 重新编译glibc?
  - 设置-mcmodel=medany，采用pc相对寻址
  - 设置-march=rv64i
  - 新的工作量，和我们的目标相违背了

不链接到glibc

- 不使用gcc，直接使用ld进行链接

```
riscv64-linux-gnu-ld: build/kernel/components/dfs/src/dfs_file.o: in function `copydir':
/home/piper/program/rt-thread/components/dfs/src/dfs_file.c:694: undefined reference to `memset'
riscv64-linux-gnu-ld: /home/piper/program/rt-thread/components/dfs/src/dfs_file.c:702: undefined reference to `strcmp'
```

- 自己实现库函数?
  - 怎么保证正确性
- kservice —— rt-thread给我们的礼物
  - 保证内核能够独立运行的一套小型的类似 C 库的函数实现子集

```
void *rt_memmove(void *dest, const void *src, rt_ubase_t n);
rt_int32_t rt_memcmp(const void *cs, const void *ct, rt_ubase_t count);
char *rt_strstr(const char *str1, const char *str2);
rt_int32_t rt_strcasecmp(const char *a, const char *b);
char *rt_strncpy(char *dest, const char *src, rt_ubase_t n);
```

封装一下就好了

- 使用kservice中的函数实现相应库函数

```
int strcmp(const char* s1, const char* s2) {
  return rt_strcmp(s1, s2);
}

int strncmp(const char* s1, const char* s2, size_t n) {
  return rt_strncmp(s1, s2, n);
}

void* memset(void* v,int c,size_t n) {
  return rt_memset(v, c, n);
}
```

- 在不链接glibc的情况下成功编译了rt-thread
  - 解决了rt-thread链接脚本带来的问题

去除压缩指令(1)

- -march
  - 控制指令集，允许编译器从指定指令集中生成指令
  - 如rv32i, rv64i, rv64imafdc...

去除压缩指令(2)

- -mabi：
  - 指定整数和浮点调用约定，由整数abi+浮点数abi组成
  - 整数abi:
    - ilp32: int, long, 指针都是32位
    - lp64: long, 指针是64位
  - 浮点abi:
    - “”：不支持浮点数在寄存器中传递
    - “f”：支持32位或者更小的浮点数
    - “d”：支持64位或更小的浮点数
  - 如ilp32, lp64f...

去除压缩指令(3)

- 设置编译选项-march=rv64imfd -mabi=lp64d
  - scons编译
  - 由于我们目前的rt-thread内核并没有使用浮点数，所以不会产生浮点相关指令
- 为什么不设置-march=rv64im -mabi=lp64
  - 默认安装的工具链编译选项为-march=rv64imafdc -mabi=lp64d
  - 没有生成lp64需要的头文件
- 一种解决方法

```
1  cd /usr/riscv64-linux-gnu/include/gnu
2  cp stubs-lp64d.h stubs-lp64.h
```

  - 这样就可以使用-march=rv64im -mabi=lp64

去除乘除指令(1)

- 设置-march=rv64ifd -mabi=lp64d

```
/home/piper/program/rt-thread/libcpu/risc-v/virt64/interrupt_gcc.S: Assembler messages:
/home/piper/program/rt-thread/libcpu/risc-v/virt64/interrupt_gcc.S:111: Error: unrecognized opcode `mul t1,t1,t2'
scons: *** [/home/piper/program/rt-thread/libcpu/risc-v/virt64/interrupt_gcc.o] Error 1
```

  - 在中断处理程序中使用汇编写了一条乘法指令

```
la    sp, __stack_start__
addi  t1, t0, 1
li    t2, __STACKSIZE__
mul   t1, t1, t2
```

- 用加法指令实现：一个偷懒的做法
  - 并没有实现完整的乘法
  - 通过上下文得出：t1的值为很小的正数

```
+     /*mul   t1, t1, t2*/
+     li    t3, 0
+ mul_begin:
+     add   t3, t3, t2
+     addi  t1, t1, -1
+     bnez  t1, mul_begin
+     mv    t1, t3
```

去除乘除指令(2)

- 新的问题

```
riscv64-linux-gnu-ld: build/kernel/src/kservice.o: in function `divide':
/home/piper/program/rt-thread/src/kservice.c:583: undefined reference to `__umoddi3'
riscv64-linux-gnu-ld: /home/piper/program/rt-thread/src/kservice.c:584: undefined reference to `__udivdi3'
riscv64-linux-gnu-ld: build/kernel/src/mem.o: in function `rt_calloc':
/home/piper/program/rt-thread/src/mem.c:512: undefined reference to `__muldi3'
riscv64-linux-gnu-ld: /home/piper/program/rt-thread/src/mem.c:540: undefined reference to `__muldi3'
```

  - __umoddi3是什么？ -> 来看看kservice.c

```
583         res = (int)(((unsigned long)*n) % 10U);
584         *n = (long)(((unsigned long)*n) / 10U);
```

- libgcc中通过软件模拟的算术库
  - long __muldi3 (long a, long b)
  - 返回a和b的乘法运算结果
- 直接从libgcc中复制一份实现就好了
  - https://github.com/riscv/riscv-gcc/blob/5964b5cd72721186ea2195a7be8d40cfe6554023/libgcc/config/riscv/div.S

```
FUNC_BEGIN (__muldi3)
  mv      a2, a0
  li      a0, 0
.L1:
  andi    a3, a1, 1
  beqz    a3, .L2
  add     a0, a0, a2
.L2:
  srli    a1, a1, 1
  slli    a2, a2, 1
  bnez    a1, .L1
  ret
FUNC_END (__muldi3)
```

跑起来了

- 获得了只包含RV64I指令的rt-thread
  - 可以在qemu上运行

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ sh qemu-nographic.sh
heap: [0x8002b388 - 0x8642b388]

 \ | /
- RT -     Thread Operating System
 / | \     4.0.4 build Aug 16 2021
 2006 - 2021 Copyright by rt-thread team
Hello RISC-V!
msh />echo "hello from rt-thread"
hello from rt-thread
msh />
```

- 在我们的cpu上运行?
  - rt-thread默认采用中断模式接收和发送串口数据

提纲

- 移植内容&使用方法
- RT-Thread文件组织
- 移植到GNU工具链
- 将中断改为轮询

RT-Thread中断模式(1)

- rt-thread在初始化以及执行完应用程序之后，内部存在两个线程

```
piper@piper:~/program/rt-thread/bsp/qemu-riscv-virt64$ sh qemu-nographic.sh
heap: [0x8002b38c - 0x8642b38c]

 \ | /
- RT -     Thread Operating System
 / | \     4.0.4 build Aug 17 2021
 2006 - 2021 Copyright by rt-thread team
Hello RISC-V!
msh />list_thread
thread               pri  status      sp     stack size max used left tick  error
-------------------- ---  ------- ---------- ---------- ------  ---------- ---
tshell                20  running 0x00000218 0x00001000    27%   0x0000000a 000
tidle0                31  ready   0x00000168 0x00004000    02%   0x00000008 000
timer                  4  suspend 0x00000178 0x00004000    02%   0x0000000a 000
msh />
```

  - tshell：rt-thread的命令行shell线程
  - tidle0：优先级最低的线程，保证内核一直有线程在运行

RT-Thread中断模式(2)

- tshell线程没有读取到串口的输入时，会将该线程挂起
  - 设置tshell线程状态为suspend（P操作）
  - 切换线程（切换到tidle0）

```
static int finsh_getchar(void)
{
    char ch = 0;
    RT_ASSERT(shell != RT_NULL);
    while (rt_device_read(shell->device, -1, &ch, 1) != 1)
        rt_sem_take(&shell->rx_sem, RT_WAITING_FOREVER);

    return (int)ch;
}
```

- 串口设备(uart)接收到字符输入就会发送一个中断信号
  - CPU获取中断号后，设置相关csr寄存器，跳转到中断处理程序执行(mtvec)

```
la t0, trap_entry
csrw SRC_XTVEC, t0              # set Trap Vector Base Address Register
```

  - 中断处理程序调用设备对应的中断服务例程
    - 在中断服务例程中执行V操作，唤醒tshell线程

将中断模式改为轮询模式(1)

- rt-thread已经实现了轮询模式,不需要进行太多的处理

```
    uart->hw_base   = UART_BASE;
    uart->irqno     = UART0_IRQ;

    virt_uart_init();

    rt_hw_serial_register(serial,
                          "uart",
                          RT_DEVICE_FLAG_STREAM | RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX,
                          uart);
    rt_hw_interrupt_install(uart->irqno, rt_hw_uart_isr, serial, "uart");

    rt_hw_interrupt_umask(uart->irqno);
```

  - rt_hw_interrupt_install: 为irqno注册一个中断服务例程
  - rt_hw_interrupt_umask:让中断控制器不要屏蔽中断号为irqno的中断
- 注释掉这两个函数
  - 其实只需要注释掉第二个就可以了

将中断模式改为轮询模式(2)

- 关闭串口的中断接收模式
  - 修改标志位

```
rt_hw_serial_register(serial,
                      "uart",
                      RT_DEVICE_FLAG_STREAM | RT_DEVICE_FLAG_RDWR, // | RT_DEVICE_FLAG_INT_RX,
                      uart);
```

```
/* open this device and set the new device in finsh shell */
if (rt_device_open(dev, RT_DEVICE_OFLAG_RDWR | RT_DEVICE_FLAG_STREAM /*| RT_DEVICE_FLAG_INT_RX */) == RT_EOK)
```

- 在shell线程没有读取到字符时不需要挂起线程
  - 让线程休眠一段时间

```
static int finsh_getchar(void)
{
    char ch = 0;
    RT_ASSERT(shell != RT_NULL);
    while (rt_device_read(shell->device, -1, &ch, 1) != 1);
        // rt_sem_take(&shell->rx_sem, RT_WAITING_FOREVER);
        rt_thread_mdelay(20);

    return (int)ch;
}
```

终于可以运行了

- 实现了相关csr和时间中断,并模拟uart之后

```
piper@piper:~/riscv64-processor$ make difftest
verilator --cc -LDFLAGS "-ldl" ./build/SimCore.v --trace --exe ./difftest/difftest.cpp
make[1]: Entering directory '/home/piper/riscv64-processor/obj_dir'
make[1]: 'VSimCore' is up to date.
make[1]: Leaving directory '/home/piper/riscv64-processor/obj_dir'
./obj_dir/VSimCore
Load ../program/rt-thread/bsp/qemu-riscv-virt64/rtthread.bin...
load program size: 19ee1
after finishlization
heap: [0x8002b1cc - 0x8642b1cc]

 \ | /
- RT -     Thread Operating System
 / | \     4.0.4 build Jul 11 2021
 2006 - 2021 Copyright by rt-thread team
Hello RISC-V!
msh />ls
No such directory
msh />echo "hello from rt-thread!"
hello from rt-thread!
msh />
```

小结

- RT-Thread的移植
  - 更换工具链:解决宏定义与头文件的相关问题
  - 去除压缩指令:修改编译选项,利用rt-thread内核库实现C库
  - 去除乘除指令:使用软件模拟的算术库
- 将中断改为轮询
  - 在初始化设备时不注册中断
  - 在没有读取到字符时不挂起线程
- 如果感兴趣的话也可以自己试试