字符读入与应用名称读取 | TimerのBlog

yanglianoo.github.io/2023/09/28/字符读入与内核栈调整

2023年9月28日

1. sys_read的实现

在之前借助Opensbi可以向串口输出一个字符，这里同理，我们可以借助Opensbi读入从串口输入来的数据，在sbi.c中新增一个获取读入串口字符的函数：

c

```
/**
 * sbi_console_getchar() - Reads a byte from console device.
 *
 * Returns the value read from console.
 */
int sbi_console_getchar(void)
{
        struct sbiret ret;

        ret = sbi_ecall(SBI_EXT_0_1_CONSOLE_GETCHAR, 0, 0, 0, 0, 0,
0, 0);

        return ret.error;
}
```

返回的字符存储在 ret.error中,读入字符的调用号是SBI_EXT_0_1_CONSOLE_GETCHAR。

借助此函数我们就能来实现sys_read了，应用程序在用户态调用sys_read来获取一个字符，内核在接收到这来自用户态的系统调用时会进行分发，然后去调用sbi_console_getchar函数将输出的字符返回给用户态，用户态的sys_read函数定义如下，__NR_read系统调用号的值为：63

c

```
int sys_read(size_t fd ,const char* buf ,
size_t len)
{
    return syscall(__NR_read,fd,buf, len);
}
```

此函数定义在app.c中，和sys_write系统调用类似，buf用来存储从串口输入的字符，对此函数封装一下,定义一个每次从串口获取一个字符的函数：

C

```
/* 获取一个字符 */
char getchar()
{
    char data[1];

sys_read(stdin,data,1
);
    return data[0];
}
```

当内核的发现来自用户态的__NR_read系统调用时需要进行分发：

```
uint64_t __SYSCALL(size_t syscall_id, reg_t arg1, reg_t arg2, reg_t arg3) {
        switch (syscall_id)
        {

        case __NR_write:
            __sys_write(arg1, arg2, arg3);
            break;
        case __NR_read:
            __sys_read(arg1, arg2, arg3);
        case __NR_sched_yield:
            __sys_yield();
            break;
        case __NR_gettimeofday:
            return __sys_gettime();
        default:
            printk("Unsupported syscall id:%d\n",syscall_id);
            break;
        }
}
```

会去调用__sys_read函数进行处理：

c

```
void __sys_read(size_t fd, const char* data,
size_t len)
{
    if(fd == stdin)
    {
        int c ;
        assert( len == 1);
        while (1)
        {
            c = sbi_console_getchar();
            if(c != -1)
                break;
        }
        char* str = translated_byte_buffer(data ,
len);
        str[0]  = c;
    }
}
```

此函数会循环读取串口的数据，直到读到一个字符，然后这里会去调用translated_byte_buffer找到从内核传进来的buf对应的实际物理地址，然后将串口读到的字符写入此物理地址。这里和__sys_write同理，应用地址空间和内核地址空间被隔离了，要进行数据传递需要找到实际的物理地址，因此translated_byte_buffer也做了一点小小的修改：

```
char * translated_byte_buffer(const char* data , size_t len)
{
    u64  user_satp = current_user_token();
    PageTable  pt ;
    pt.root_ppn.value = MAKE_PAGETABLE(user_satp);

    u64 start_va = data;
    u64 end_va = start_va + len;
    VirtPageNum vpn = floor_virts(virt_addr_from_size_t(start_va));
    PageTableEntry* pte = find_pte(&pt , vpn);

    //拿到物理页地址
    int mask = ~( (1 << 10) -1 );
    u64 phyaddr = ( pte->bits & mask) << 2 ;
    //拿到偏移地址
    u64 page_offset = start_va & 0xFFF;
    u64 data_d = phyaddr + page_offset;
    return (char*) data_d;
}
```

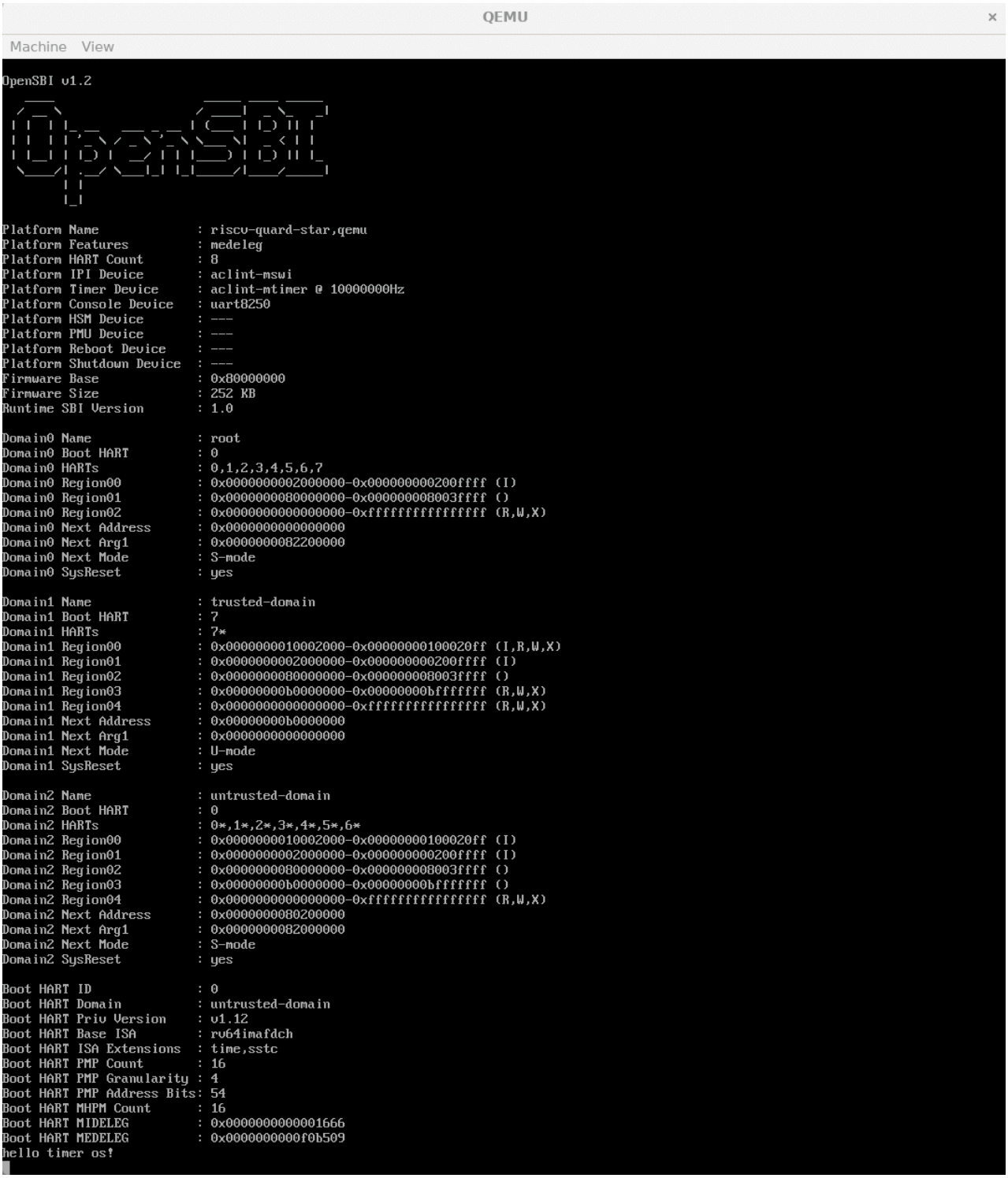
可以看见根据传入的用户态的地址转换后返回实际对应的物理地址。

来测试一下，首先修改一下应用程序，让`time`应用程序不打印东西，在`write`应用程序中来读取字符：

```c
#include <timeros/types.h>
#include <timeros/syscall.h>
#include <timeros/string.h>
int main()
{

    while (1)
    {
        char data = getchar();
        printf("%c",data);
    }
    return 0;
}
```

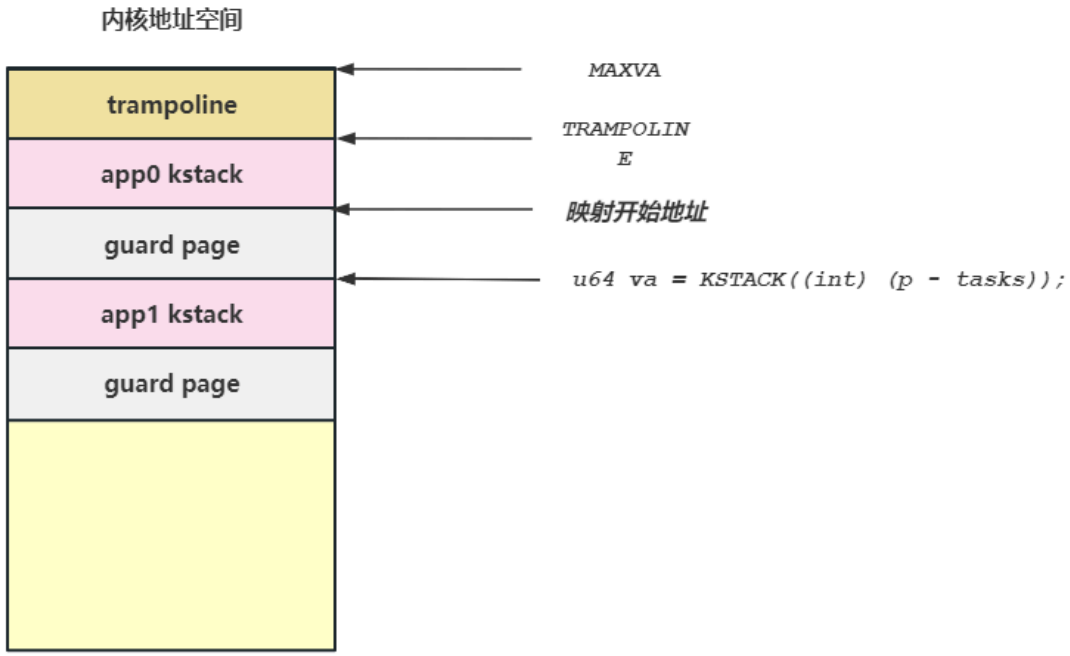
编译内核和执行：



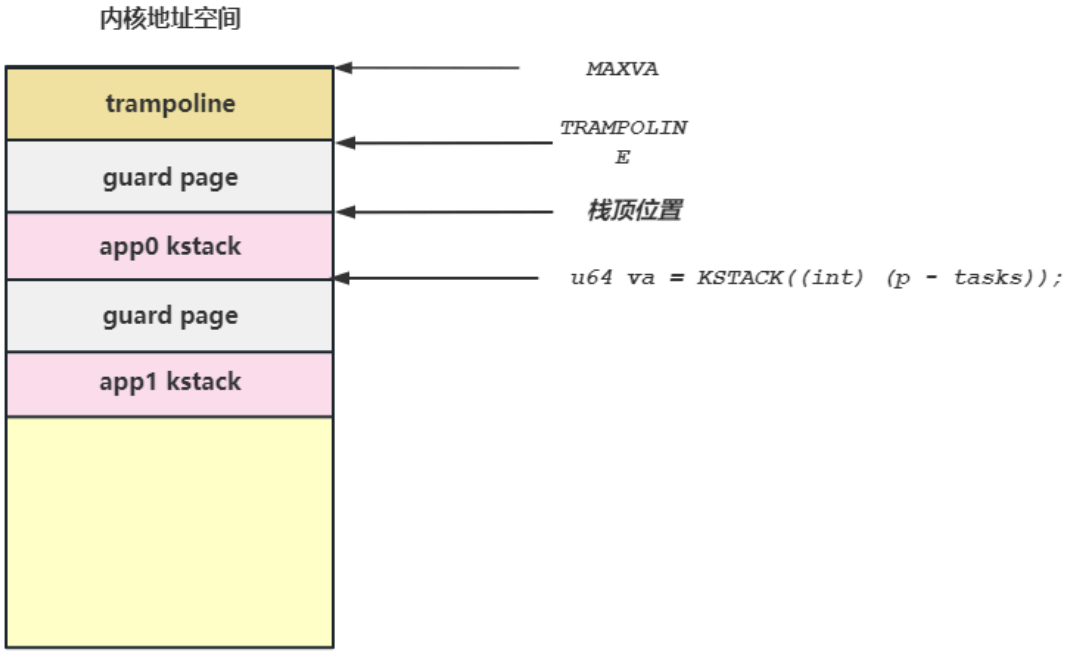
可以看见我从键盘输出的字符都成功打印了出来。

2. 内核栈的修改

在内核和用户程序的映射逻辑 | TimerのBlog (yanglianoo.github.io)我们对每个应用程序的内核栈进行了映射，当时映射后的内存分布长这样子：



这样不太好，我当时脑子抽了，实际上trampoline和app0 kstack之间应该存在一页guard page才对，修改后的内存分布长这样子：



映射内核栈的代码也要随之修改一下：

C

```c
/* 为每个应用程序映射内核栈,内核空间以及进行了映射 */
void proc_mapstacks(PageTable* kpgtbl)
{
  struct TaskControlBlock *p;

  for(p = tasks; p < &tasks[MAX_TASKS]; p++) {
    char *pa = (char*)phys_addr_from_phys_page_num(kalloc()).value;
    if(pa == 0)
      panic("kalloc");
    u64 va = KSTACK((int) (p - tasks));
    PageTable_map(kpgtbl, virt_addr_from_size_t(va ),
phys_addr_from_size_t((u64)pa), \
                  PAGE_SIZE, PTE_R | PTE_W);
    // 给应用内核栈赋值
    p->kstack = va +  PAGE_SIZE;
  }
}
```

3. 读取应用程序的名称

在后续的开发中我们会使用进程的名字来对应用程序加载和执行，因此需要内核能得到应用的名称，在我们之前的build.c中加入几行代码：

```c
    for (int i = 0; i < app_count; i++) {
        fprintf(f, "\n.quad app_%d_start", i);
    }
    fprintf(f, "\n.quad app_%d_end", app_count - 1);

    fprintf(f,"\n.global _app_names\n_app_names:");
    for (int i = 0; i < app_count; i++)
    {
        fprintf(f,"\n.string \"%s\"",apps[i]);
    }

    for (int i = 0; i < app_count; i++) {
        printf("app_%d: %s\n", i, apps[i]);
        fprintf(f, "\n.section .data\n.global app_%d_start\n.global app_%d_end\n.align 3\napp_%d_sta
        free(apps[i]);
    }

    fclose(f);
}
```

加入这几行代码后，link_app.S也会随之改变，原因在于我们按照顺序将各个应用的名字通过 .string 伪指令放到数据段中，注意链接器会自动在每个字符串的结尾加入分隔符 \0 ，它们的位置则由全局符号 _app_names 指出。

```
    .align 3
    .section .data
    .global _num_app
_num_app:
    .quad 2
    .quad app_0_start
    .quad app_1_start
    .quad app_1_end
    .global _app_names
_app_names:
    .string "time"
    .string "write"
    .section .data
    .global app_0_start
    .global app_0_end
    .align 3
app_0_start:
    .incbin "../user/bin/time"
app_0_end:
    .section .data
    .global app_1_start
    .global app_1_end
    .align 3
app_1_start:
    .incbin "../user/bin/write"
app_1_end:
```

在`loader.c`中新建一个函数来读取应用的名字：

C

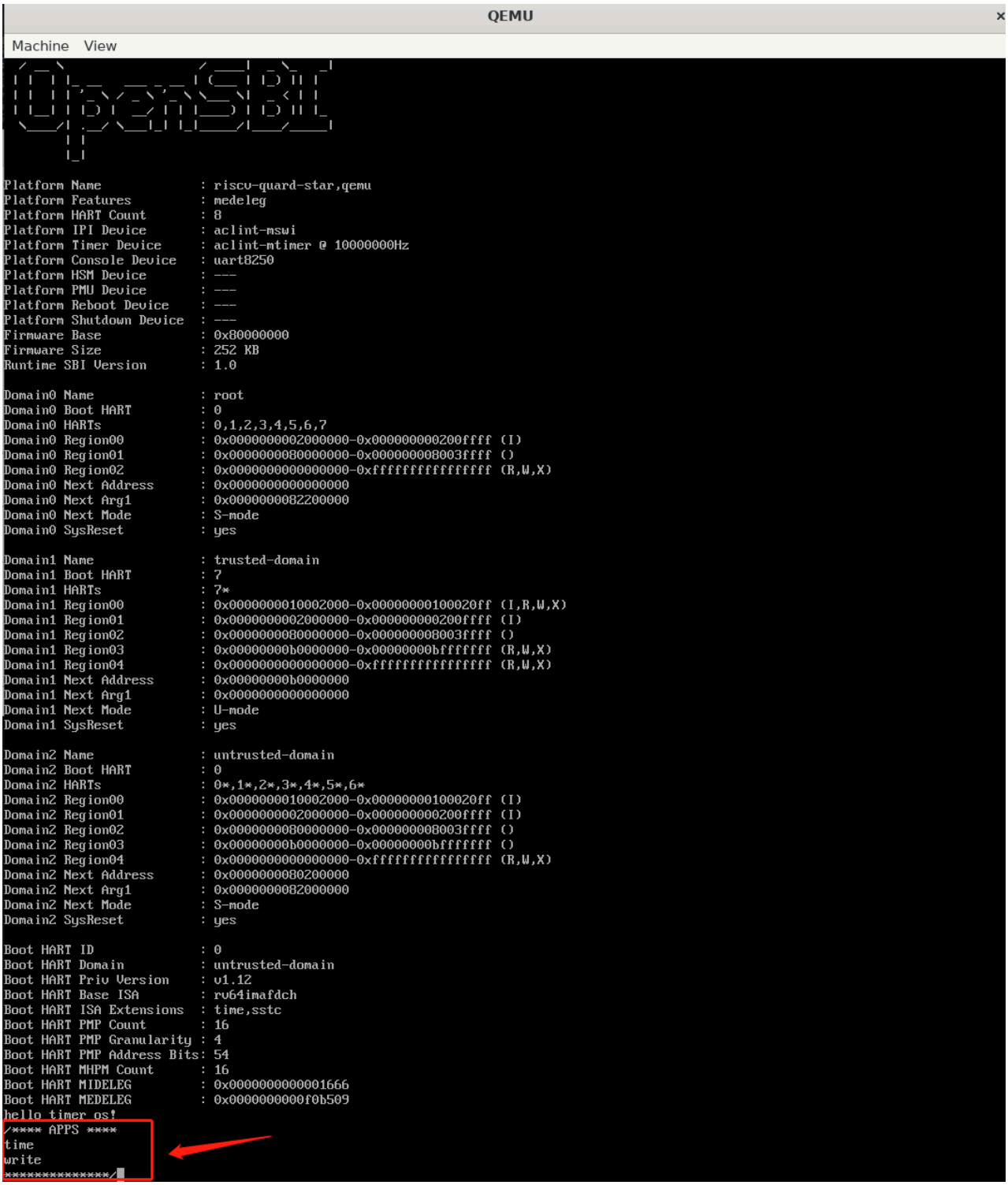
```
extern char _app_names[];
static char* app_names[MAX_TASKS];
void get_app_names()
{
    int app_num = get_num_app();
    printk("/**** APPS ****\n");
    for (size_t i = 0; i < app_num; i++)
    {
        if(i==0)
        {
            size_t len = strlen(_app_names);
            app_names[0] = _app_names;
        }
        else
        {
            size_t len = strlen(app_names[i-1]);
            app_names[i] = (char*)((u64)_app_names + i * len
+ 1);
        }

        printk("%s\n",app_names[i]);

    }
    printk("**************/");

}
```

由于链接器会自动在每个字符串的结尾加入分隔符 `\0`，因此根据`\0`将每个应用程序的名称存储到了`app_names`这个数组中。在main函数中调用此函数测试一下：



成功解析到程序的名称。

参考链接

进程管理的核心数据结构 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)