实现user_shell | TimerのBlog

yanglianoo.github.io/2023/10/26/实现user-shell

2023年10月26日

1.bug修复

在实现shell应用程序时，我发现了许多bug，在完成shell这个应用程序之前先把之前代码的bug修复了

1.1 sys_fork 错误修复

```
int __sys_fork()
{
  struct TaskControlBlock* np;
  struct TaskControlBlock* p = current_proc();
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // 拷贝父进程的内存数据，根据页表查找物理页拷贝
  uvmcopy(&p->pagetable,&np->pagetable,p->base_size);

  // 拷贝父进程的trap页数据
  memcpy((void*)np->trap_cx_ppn,(void*)p->trap_cx_ppn,PAGE_SIZE);

  // 子进程返回值为0
  TrapContext* cx_ptr = np->trap_cx_ppn;
  cx_ptr->a0 = 0;
  cx_ptr->kernel_sp = np->kstack;
  // 复制TCB的信息
  np->entry = p->entry;
  np->base_size = p->base_size;
  np->parent = p;
  np->ustack = p->ustack;

  np->task_context = tcx_init((reg_t)np->kstack);

  _top++;
  return np->pid;
}
```

- 在之前的实现中，trap页中内核栈忘记覆盖了，因为sys_fork会去空闲任务数组中拿到一个，此时内核栈和父进程不一样，因此trap页中内核栈的地址需要重新赋值：`cx_ptr->kernel_sp = np->kstack;`
- 初始化任务上下文，这里我只是用tcx_init这个函数来初始化了，之前是直接赋值，这里优化了一下

1.2 sys_exec 错误修复

c

```
int exec(const char* name)
{

    AppMetadata metadata =
get_app_data_by_name(name);
    if(metadata.id<0)
    {
      return -1;
    }
    //ELF 文件头
    elf64_ehdr_t *ehdr = metadata.start;
    elf_check(ehdr);

    struct TaskControlBlock* proc =
current_proc();
    PageTable old_pagetable = proc->pagetable;
    u64 oldsz = proc->base_size;
    //重新分配页表
    proc_pagetable(proc);
    //加载程序段
    load_segment(ehdr,proc);
    //映射应用程序用户栈开始地址
    proc_ustack(proc);

    TrapContext* cx_ptr = proc->trap_cx_ppn;
    cx_ptr->sepc = (u64)ehdr->e_entry;
    cx_ptr->sp = proc->ustack;

    proc_freepagetable(&old_pagetable,oldsz);
    return 0;
}
```

在exec实现时，我们是没有重新分配trap页，我们只是根据传进来的elf文件对程序进行覆盖替换掉原来的，因此只是重新分配了根页表，然后对地址空间重新进行了映射。本质上exec前后都还是同一个进程，只是地址空间不同了，trap页中的某些数据不同了：1. 程序入口地址e_entry 2. 用户栈地址u_stack。因此trap页中需要替换的也就是这两个部分，其他是不需要修改的，比如内核栈、trap_handler地址什么的都是没变的，不需要覆盖。同时由于trap页没有被重新分配物理页，只是改变了映射的地址空间，因此此页不能被释放掉

```
void proc_freepagetable(PageTable* pagetable, u64 sz)
{
  uvmunmap(pagetable, floor_virts(virt_addr_from_size_t(TRAMPOLINE)), 1, 0);
  uvmunmap(pagetable, floor_virts(virt_addr_from_size_t(TRAPFRAME)), 1, 0);
  uvmfree(pagetable, sz);
}
```

在proc_freepagetable中，trap页物理内存是否释放的标志位修改为0，不能被释放掉

1.3 sys_read 问题修复

```
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;
            schedule();
            continue;
        }
        char* str = translated_byte_buffer(data);
        str[0]  = c;
    }
}
```

之前的实现中，没有去调用schedule函数，这样造成的后果就是程序会阻塞在sys_read中出不去，加上调度后的逻辑就是sys_read在读取串口的字符，如果此时没读到就调度执行其他进程，重新调度回sys_read时再判断有没有读到字符，这样就不会阻塞在这里了。

2. user_shell 实现

内核在执行user_shell这个应用程序时，应该会有一个initproc的初始化进程，user_shell是由这个初始化进程fork来的，初始化进程负责拉起user_shell和回收结束执行的子进程的资源。初始化进程是内核默认加载执行的第一个应用程序。在这里我们直接sys_exec来执行user_shell，因为还没实现子进程的资源回收机制，因此initproc的程序如下：

c

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

sys_exec("user_shell");
}
```

在内核中手动拉起此初始化进程，：

c

```
//加载
进程
load_a
pp(0);
app_in
it(0);
```

接下来实现`user_shell`：

c

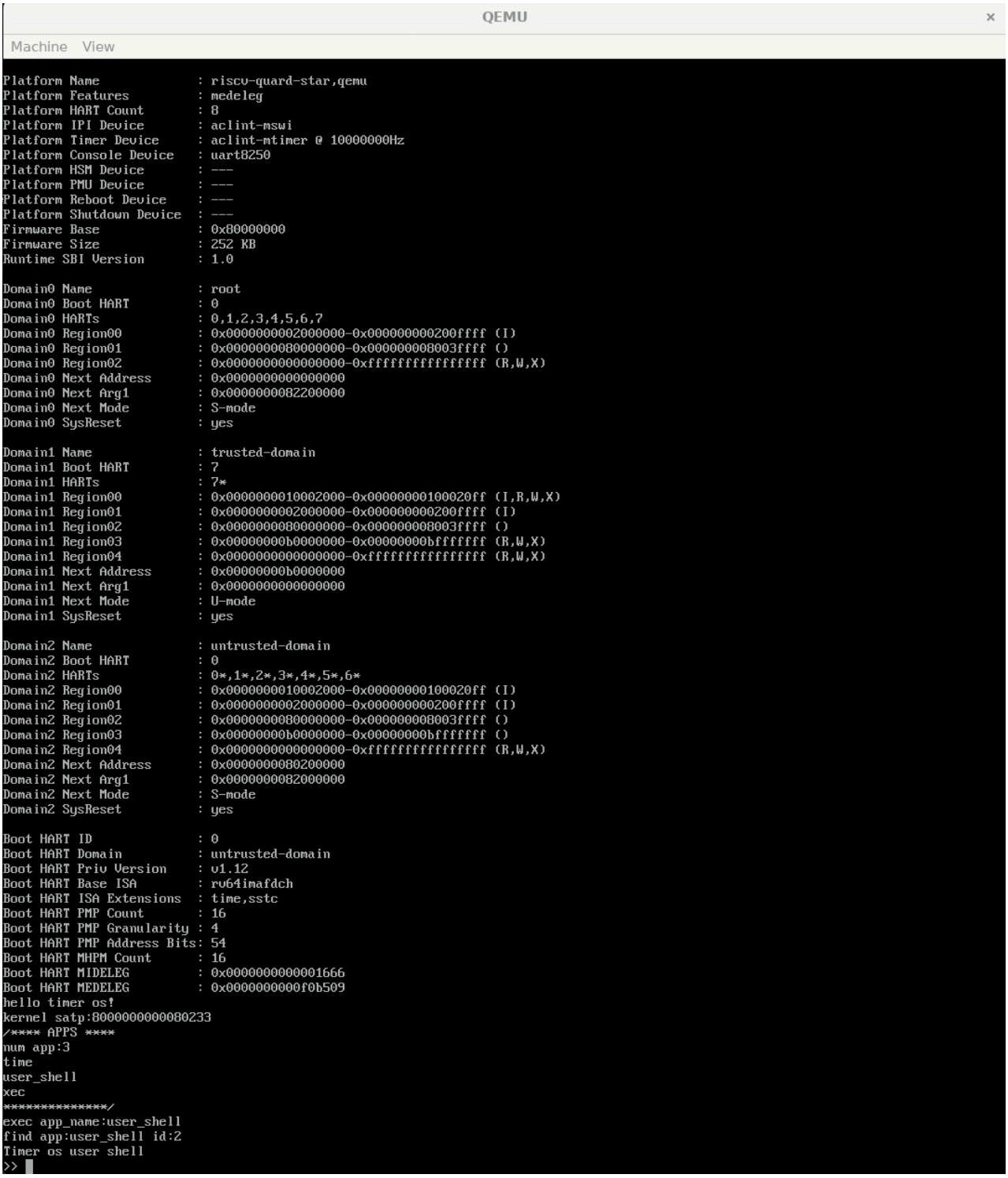
```c
#include <timeros/types.h>
#include <timeros/syscall.h>
#include <timeros/string.h>

#define LF 0x0a
#define CR 0x0d   //enter
#define DL 0x7f
#define BS 0x08   // backspace
#define BUFFER_SIZE 1024
int main()
{
    printf("Timer os user shell\n");
    printf(">> ");
    char line[BUFFER_SIZE];
    while (1)
    {
        char c = getchar();
        switch (c)
        {
        case CR:
            printf("\n");
            if(strlen(line) > 0)
            {
                line[strlen(line)] =
'\0';
                int pid = sys_fork();
                if(pid==0)
                {
                    sys_exec(line);
                }
            }
            break;
        case BS:
            if (strlen(line) > 0)
            {
                printf("\b \b");
                line[strlen(line) - 1] =
'\0';
            }
            break;
        default:
            printf("%c",c);
            strncat(line,(char*)&c,1);
            break;
        }


    }
    return 0;
}
```

- user_shell的逻辑就是从键盘获取输入，将读到的字符放进line这个字符数组中保存，如果这个字符是键盘上的enter键，则fork一个子进程，让子进程通过sys_exec来执行用户通过键盘输出的可执行程序；如果是键盘上的backspace键，意味着删除一个字符，在c语言中\b是退格字符，当它被打印时，它会导致光标向后移动一格，覆盖先前打印的字符。具体而言，printf("\b \b"); 会打印一个退格字符，然后打印一个空格，最后再打印一个退格字符。这会导致光标向后移动一格，然后再向前移动一格，从而导致看起来好像什么都没有打印一样。
- 这里用到了一个函数strncat，这个函数的作用就是用于向一个字符串的末尾拼接字符，具体实现如下：

c

```
void strncat(char *dest, const char *src,
int n) {
    while (*dest) {
        dest++;
    }
    while (n > 0 && *src) {
        *dest++ = *src++;
        n--;
    }
    *dest = '\0';
}
```

3.测试



在上面的测试中，可以看见user_shell由initproc拉起，然后我测试了一下字符输入与删除，接着输出xec然后按enter键，此时会去执行xec这个子进程，此进程会循环打印exec!，测试成功!

4. 参考链接

- 进程概念及重要系统调用 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)
- C语言中的转义字符\b的含义_c语言\b-CSDN博客

文章作者: Timer
文章链接: https://yanglianoo.github.io/2023/10/26/实现user-shell/

相关推荐