

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:

```

#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

```
QEMU x
Machine View

Platform Name      : riscu-guard-star,qemu
Platform Features  : medeleg
Platform HART Count : 8
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : ---
Platform Shutdown Device : ---
Firmware Base      : 0x80000000
Firmware Size       : 252 KB
Runtime SBI Version : 1.0

Domain0 Name       : root
Domain0 Boot HART  : 0
Domain0 HARTs       : 0,1,2,3,4,5,6,7
Domain0 Region00    : 0x0000000002000000-0x000000000200ffff (I)
Domain0 Region01    : 0x0000000008000000-0x0000000008003ffff ( )
Domain0 Region02    : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address : 0x0000000000000000
Domain0 Next Arg1    : 0x00000000082200000
Domain0 Next Mode    : S-mode
Domain0 SysReset     : yes

Domain1 Name       : trusted-domain
Domain1 Boot HART  : 7
Domain1 HARTs       : 7*
Domain1 Region00    : 0x00000000010002000-0x000000000100020fff (I,R,W,X)
Domain1 Region01    : 0x0000000002000000-0x000000000200ffff (I)
Domain1 Region02    : 0x0000000008000000-0x0000000008003ffff ( )
Domain1 Region03    : 0x000000000b0000000-0x000000000bffffff (R,W,X)
Domain1 Region04    : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain1 Next Address : 0x000000000b0000000
Domain1 Next Arg1    : 0x0000000000000000
Domain1 Next Mode    : U-mode
Domain1 SysReset     : yes

Domain2 Name       : untrusted-domain
Domain2 Boot HART  : 0
Domain2 HARTs       : 0*,1*,2*,3*,4*,5*,6*
Domain2 Region00    : 0x00000000010002000-0x000000000100020fff (I)
Domain2 Region01    : 0x0000000002000000-0x000000000200ffff (I)
Domain2 Region02    : 0x0000000008000000-0x0000000008003ffff ( )
Domain2 Region03    : 0x000000000b0000000-0x000000000bffffff ( )
Domain2 Region04    : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain2 Next Address : 0x00000000080200000
Domain2 Next Arg1    : 0x00000000082000000
Domain2 Next Mode    : S-mode
Domain2 SysReset     : yes

Boot HART ID       : 0
Boot HART Domain   : untrusted-domain
Boot HART Priv Version : v1.12
Boot HART Base ISA  : rv64imafdc
Boot HART ISA Extensions : time,sstc
Boot HART PMP Count : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits : 54
Boot HART MHPM Count : 16
Boot HART MIDELEG   : 0x0000000000001666
Boot HART MEDELEG   : 0x0000000000f0b509
hello timer os!
kernel satp:8000000000080233
/**** APPS ****
num app:3
time
user_shell
xec
*****/
exec app_name:user_shell
find app:user_shell id:2
Timer os user_shell
>>
```

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/>

版权声明: 本博客所有文章除特别声明外, 均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

相关推荐