

1.exec系统调用原型

在linux系统下系统调用exec是以新的进程去代替原来的进程，但进程的PID保持不变。因此，可以这样认为，exec系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。在我们的内核中，如果仅有fork的话，那么所有的进程都只能和用户初始进程一样执行同样的代码段，这显然是远远不够的。于是我们还需要引入exec系统调用来执行不同的可执行文件：

C

```
#define __NR_execve 221
int sys_exec(char* name)
{
    return
    syscall(__NR_execve,0,name,0);
}
```

上面这段代码是定义在用户态的app.c中，name为传递给sys_exec函数的参数，代表了要加载的可执行文件的名称。实际上一般来说可执行文件都有参数，因此参数也需要通过sys_exec传递进去，这里先默认无参数。

用户在用户态调用此函数来加载执行一个可执行文件，内核通过系统调用分发，因此在内核的sys_call.c中做了如下修改：

C

```
uint64_t __sys_exec(const char* name)
{
    char* app_name =
    translated_byte_buffer(name);
    printk("exec app_name:%s\n",app_name);
    exec(app_name);
    return 0;
}
```

```

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();
        case __NR_clone:
            return sys_fork();
        case __NR_execve:
            return __sys_exec(arg2);
        default:
            printk("Unsupported syscall id:%d\n", syscall_id);
            break;
    }
}

```

和之前同理用户态传进来的字符串的参数由于地址空间的不同因此需要先去调用 `translated_byte_buffer` 转换一下，然后去调用 `exec` 函数，此函数会用来根据传入的 `app` 的名字进行加载执行

在实现 `exec` 函数之前，先来精简一下 `loader.c` 中的代码：

C

```

void load_app(size_t app_id)
{
    //加载ELF文件
    AppMetadata metadata = get_app_data(app_id
+ 1);
    //ELF 文件头
    elf64_ehdr_t *ehdr = metadata.start;
    //检查elf 文件
    elf_check(ehdr);
    //创建任务
    TaskControlBlock* proc =
task_create_pt(app_id);
    //加载程序段
    load_segment(ehdr, proc);
    //赋值任务的 entry
    proc->entry = (u64)ehdr->e_entry;
    // 映射应用程序用户栈开始地址
    proc_ustack(proc);
}

```

对 `load_app` 此函数进行了简化，封装了三个函数，一个是 `elf_check(elf64_ehdr_t *ehdr)`，用于检查传入的 `elf` 文件的魔数，另一个是 `load_segment(elf64_ehdr_t *ehdr, struct TaskControlBlock* proc)`，用于加载应用程序的数据并映射，第三个是 `proc_ustack(struct TaskControlBlock *p)`，用于映射应用程序的用户栈。

C

```
void elf_check(elf64_ehdr_t *ehdr)
{
    //判断 elf 文件的魔数
    assert(*(u32 *)ehdr==ELFMAG);
    //判断传入文件是否为 riscv64 的
    if (ehdr->e_machine != EM_RISCV || ehdr->e_ident[EI_CLASS] !=
    ELFCLASS64)
    {
        panic("only riscv64 elf file is supported");
    }
}
```

C

```
void load_segment(elf64_ehdr_t *ehdr,struct TaskControlBlock* proc)
{
    elf64_phdr_t *phdr;
    for (size_t i = 0; i < ehdr->e_phnum; i++)
    {
        //拿到每个Program Header的指针
        phdr =(u64) (ehdr->e_phoff + ehdr->e_phentsize * i + (u64)ehdr);
        if(phdr->p_type == PT_LOAD)
        {
            // 获取映射内存段开始位置
            u64 start_va = phdr->p_vaddr;
            // 获取映射内存段结束位置
            proc->ustack = start_va + phdr->p_memsz;
            // 转换elf的可读, 可写, 可执行的 flags
            u8 map_perm = PTE_U | flags_to_mmap_prot(phdr->p_flags);
            // 获取映射内存大小,需要向上对齐
            u64 map_size = PGROUNDUP(phdr->p_memsz);
            for (size_t j = 0; j < map_size; j+= PAGE_SIZE)
            {
                // 分配物理内存, 加载程序段, 然后映射
                PhysPageNum ppn = kalloc();
                //获取到分配的物理内存的地址
                u64 paddr = phys_addr_from_phys_page_num(ppn).value;
                memcpy(paddr, (u64)ehdr + phdr->p_offset + j, PAGE_SIZE);
                //内存逻辑段内存映射
                PageTable_map(&proc->pagetable,virt_addr_from_size_t(start_va +
j), \
                                phys_addr_from_size_t(paddr), PAGE_SIZE ,
map_perm);
            }
        }
    }
    proc->ustack = 2 * PAGE_SIZE + PGROUNDUP(proc->ustack);
    proc->base_size=proc->ustack;
}
```

```

void proc_ustack(struct TaskControlBlock *p)
{
    // 映射应用程序用户栈开始地址
    PhysPageNum ppn = kalloc();
    u64 paddr = phys_addr_from_phys_page_num(ppn).value;
    PageTable_map(&p->pagetable, virt_addr_from_size_t(p->ustack -
    PAGE_SIZE), phys_addr_from_size_t(paddr), \
                PAGE_SIZE, PTE_R | PTE_W | PTE_U);
}


```

然后是修复一个bug:

```

/* 根据app的名字返回app的id */
AppMetadata get_app_data_by_name(char* path)
{
    AppMetadata metadata;
    int app_num = get_num_app();
    for (size_t i = 0; i < app_num; i++)
    {
        if(strcmp(path, app_names[i]) == 0)
        {
            metadata = get_app_data(i+1);
            printk("find app:%s id:%d\n", path, metadata.id);
            return metadata;
        }
    }
    printk("not exit!!\n");
}

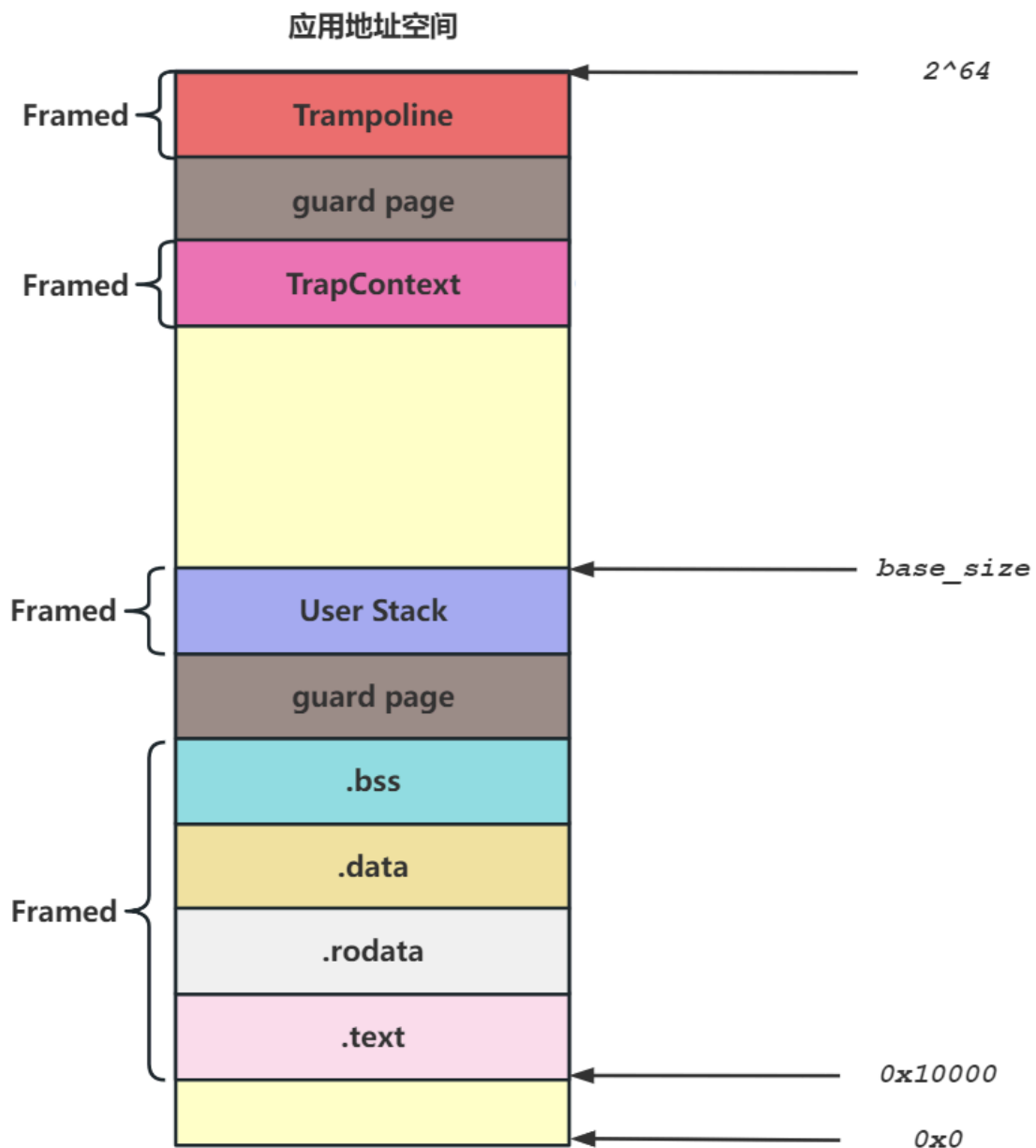
```



这个根据程序名获取应用程序的函数，这里应该是`get_app_data(i)`。

2. 解除映射与释放内存

一个进程通过`exec`系统调用来加载另外一个应用程序来进行执行，那么在这个新的应用程序开始执行的时候，原有进程的地址空间生命周期就可以结束了，里面包含的全部物理页帧都会被回收，新的应用程序需要为其分配新的页表，然后映射新的地址空间。首先同样回忆一下一个应用程序包含了哪些内存地址空间：



每个应用程序依次映射了：跳板页，`trap`上下文页，用户栈页，应用程序页。销毁一个应用程序的地址空间首先就是要解除映射关系，然后将内核分配给此应用程序的物理内存释放掉。由于所有程序的跳板页都是映射到同一页物理地址，这一页是不能是不能释放物理内存的，只需要解除映射关系即可。

在`address.c`中定义了一个函数用于销毁一个应用程序的地址空间：

c

```
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, 1);
    uvmfree(pagetable, sz);
}
```

此函数的参数为应用程序的根页表和应用程序的大小`base_size`，此函数首先是

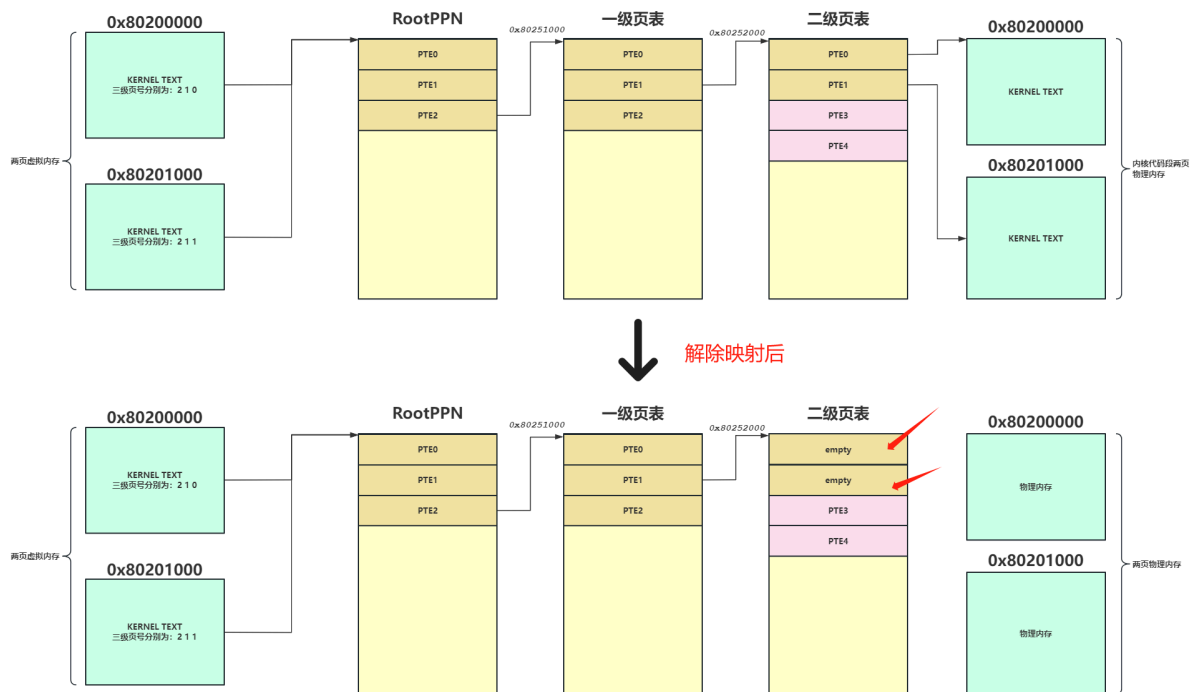
- 调用`uvmunmap`函数将`TRAMPOLINE`页解除了映射关系，并未释放此页内存
- 调用`uvmunmap`函数将`TRAPFRAME`页解除了映射关系，并且释放了此页内存
- 调用`uvmfree`函数将应用程序从`0x10000`开始到`base_size`之间的内存页解除了映射关系，并且释放掉了对应的物理内存。释放掉了页表所占的三页内存

首先来看`uvmunmap`函数：

C

```
/* 取消映射 */
void uvmunmap(PageTable* pt, VirtPageNum vpn, u64 npages, int do_free)
{
    PageTableEntry* pte;
    u64 a;
    for (a = vpn.value; a < vpn.value + npages; a++)
    {
        pte = find_pte(pt, virt_page_num_from_size_t(a));
        if(pte !=0 )
        {
            if(do_free)
            {
                u64 phyaddr = PTE2PA(pte->bits);
                PhysPageNum ppn =
                floor_phys(phys_addr_from_size_t(phyaddr));
                kfree(ppn);
            }
            *pte = PageTableEntry_empty();
        }
    }
}
```

- 传入参数为应用程序的根页表，需要取消映射的虚拟地址的起始地址，需要取消映射的页数，是否释放内存的标志位
- 从`vpn`开始，通过`find_pte`函数去查看此页是否被映射，如果被映射了，则取消映射即将第三级的页表项赋值为空
- 如果`do_free`成立，则去释放掉对应的物理内存

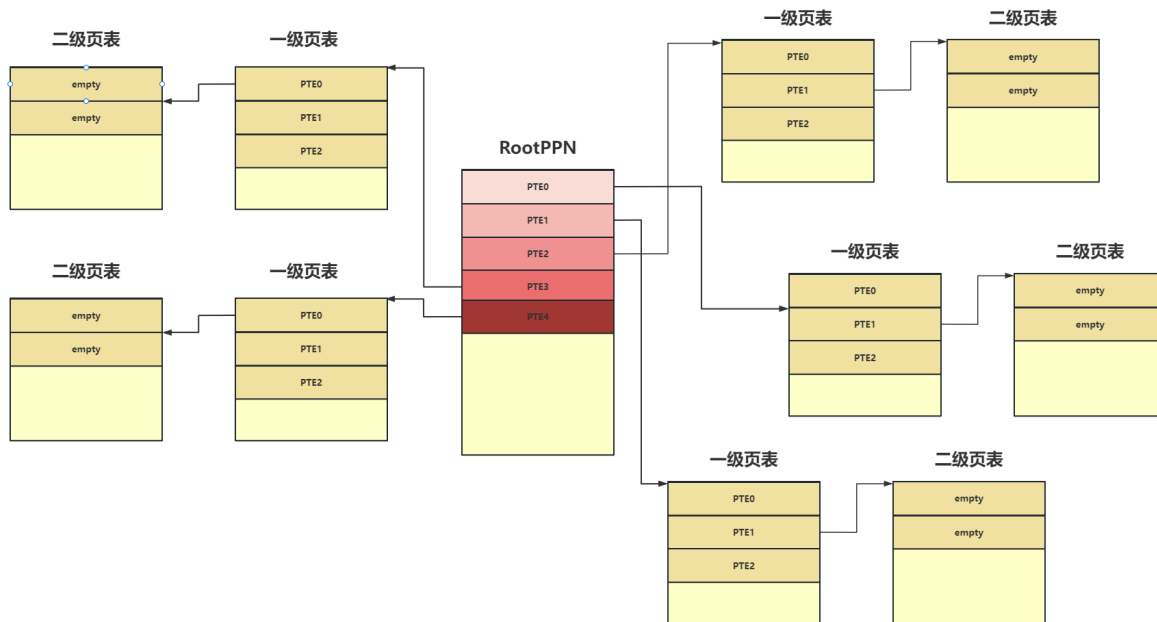


然后来看uvmfree函数:

C

```
void uvmfree(PageTable* pt , u64 sz)
{
    if(sz > 0)
    {
        uvmunmap(pt, floor_virts(virt_addr_from_size_t(0)), sz/PAGE_SIZE, 1);
    }
    freewalk(pt->root_ppn);
}
```

- 首先是调用uvmunmap函数, 将应用程序虚拟地址从0~base_sz之间的的映射关系取消掉, 同时释放掉对应的物理内存
- 然后是调用freewalk函数将此地址空间页表占用的物理空间全部释放掉



从根页表开始映射，一个应用程序的映射关系如上图所示，每个页表页有512个页表项，因此需要递归搜寻去释放内存：

C

```
/* 解除页表映射关系，释放内存*/
void freewalk(PhysPageNum ppn)
{
    for (int i = 0; i < 512; i++)
    {
        PageTableEntry* pte = &get_pte_array(ppn)[i];
        if((pte->bits & PTE_V) && (pte->bits & (PTE_R|PTE_W|PTE_X))
== 0)
        {
            PhysPageNum child_ppn = PageTableEntry_ppn(pte);
            freewalk(child_ppn);
            *pte = PageTableEntry_empty();
        }
        else if(pte->bits & PTE_V)
        {
            panic("freewalk: leaf");
        }
    }
    kfree(ppn);
}
```

由于freewalk函数是在解除映射关系后才调用的，因此在上述的映射关系中，二级页表的所有页表项都是空的，if((pte->bits & PTE_V) && (pte->bits & (PTE_R|PTE_W|PTE_X)) == 0)这个判断条件只在根页表和一级页表才成立，此时才会去向下搜寻下一级页表知道三级页表搜寻完毕。

3.exec系统调用的实现

C

```
void exec(const char* name)
{
    AppMetadata metadata =
get_app_data_by_name(name);
    //ELF 文件头
    elf64_ehdr_t *ehdr = metadata.start;
    elf_check(ehdr);

    struct TaskControlBlock* proc =
current_proc();
    //保存旧的页表
    PageTable old_pagetable = proc->pagetable;
    //拿到旧进程的数据大小
    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;
    reg_t sstatus = r_sstatus();
    // 设置 sstatus 寄存器第8位即SPP位为0 表示为
U模式
    sstatus &= (0U << 8);
    w_sstatus(sstatus);
    cx_ptr->sstatus = sstatus;
    // 设置内核页表token
    cx_ptr->kernel_satp = kernel_satp;
    // 设置内核栈虚拟地址
    cx_ptr->kernel_sp = proc->kstack;
    // 设置内核trap_handler的地址
    cx_ptr->trap_handler = (u64)trap_handler;

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

- 根据传入的应用程序的名字拿到此程序的应用数据
- 调用`proc_pagetable`函数为新的应用程序创建一个空的用户的页表，映射跳板页，映射用户程序的`trap`页
- 为新的应用程序加载和映射程序段，映射用户栈
- 填充`trap`页

- 解除旧进程的映射和释放物理内存

这里无需对任务上下文进行处理，因为这个进程本身已经在执行了，而只有被暂停的应用才需要在内核栈上保留一个任务上下文。

4. 系统调用后重新获取 Trap 上下文

过去的 `trap_handler` 实现是这样处理系统调用的：

```
void trap_handler()
{
    set_kernel_trap_entry();
    TrapContext* cx = get_current_trap_cx();
    reg_t scause = r_scause();
    reg_t cause_code = scause & 0xffff;
    if(scause & 0x8000000000000000)
    {
        switch (cause_code)
        {
            /* rtc 中断*/
            case 5:
                set_next_trigger();
                schedule();
                break;
            default:
                printk("undfined interrrupt scause:%x\n",scause);
                break;
        }
    }
    else
    {
        switch (cause_code)
        {
            /* U模式下的syscall */
            case 8:
                cx->sepc += 8;
                cx->a0 = __SYSCALL(cx->a7,cx->a0,cx->a1,cx->a2);
                break;
            default:
                printk("undfined exception scause:%x\n",scause);
                break;
        }
    }
    trap_return();
}
```

这里的 `cx` 是当前应用的 Trap 上下文的指针，我们需要通过查页表找到它具体被放在哪个物理页帧上，并构造相同的虚拟地址来在内核中访问它。对于系统调用 `sys_exec` 来说，一旦调用它之后，我们会发现 `trap_handler` 原来上下文中的 `cx` 失效了——因为它是用来访问之前地址空间中 Trap 上下文被保存在的那个物理页帧的，而现在它已经被回收掉了。因此，为了能够处理类似的这种情况，我们在 `syscall` 分发函数返回之后需要重新获取 `cx`，目前的实现如下：

```

void trap_handler()
{
    set_kernel_trap_entry();
    TrapContext* cx = get_current_trap_cx();
    reg_t scause = r_scause();
    reg_t cause_code = scause & 0xffff;
    if(scause & 0x8000000000000000)
    {
        switch (cause_code)
        {
            /* rtc 中断*/
            case 5:
                set_next_trigger();
                schedule();
                break;
            default:
                printk("undfined interrrupt scause:%x\n",scause);
                break;
        }
    }
    else
    {
        switch (cause_code)
        {
            /* U模式下的syscall */
            case 8:
                cx->sepc += 8;
                u64 result = __SYSCALL(cx->a7,cx->a0,cx->a1,cx->a2);
                cx = get_current_trap_cx();
                cx->a0 = result;
                break;
            default:
                printk("undfined exception scause:%x\n",scause);
                break;
        }
    }
    trap_return();
}

```

5. 测试

在user目录下新建一个应用程序xec.c:

C

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

int main()
{
    while (1)
    {

printf("exec!\n");
    }
    return 0;
}
```

我们让write.c通过sys_exec系统调用来执行此程序：

C

```
#include
<timeros/types.h>
#include
<timeros/syscall.h>
#include
<timeros/string.h>
int main()
{
    sys_exec("xec");
    return 0;
}
```

我修改了一下user目录下的Makefile使得通过一个make命令就可编译所用的应用程序

makefile

```
CROSS_COMPILE = riscv64-unknown-elf-
CFLAGS = -nostdlib -fno-builtin -mmodel=medany

CC = ${CROSS_COMPILE}gcc
OBJCOPY = ${CROSS_COMPILE}objcopy
OBJDUMP = ${CROSS_COMPILE}objdump
INCLUDE:=-I../include

LIB = ../lib

all: time write xec

write: write.c $(LIB)/*.c
    ${CC} ${CFLAGS} $(INCLUDE) -T user.ld -Wl,-Map=write.map -o
bin/write $^

time: time.c $(LIB)/*.c
    ${CC} ${CFLAGS} $(INCLUDE) -T user.ld -Wl,-Map=time.map -o
bin/time $^

xec: xec.c $(LIB)/*.c
    ${CC} ${CFLAGS} $(INCLUDE) -T user.ld -o bin/xec $^

debug: objdump_time objdump_write objdump_xec

objdump_time:
    ${OBJDUMP} -d bin/time > time.txt
objdump_write:
    ${OBJDUMP} -d bin/write > write.txt
objdump_xec:
    ${OBJDUMP} -d bin/xec > xec.txt
```

编译运行，现在os/user目录执行make命令生成应用程序，然后到quard-star目录编译运行，结果如下：



可以看见xec被成功执行了！！

参考链接

进程管理机制的设计实现 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)

文章作者: Timer

文章链接: <https://yanglianoo.github.io/2023/10/20/sys-exec的实现/>

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