基于地址空间的分时多任务 | TimerのBlog

yanglianoo.github.io/2023/09/13/基于地址空间的分时多任务

2023年9月13日

目标

这一节的目标是实现基于地址空间的分时多任务，内核和应用之间的地址空间是隔离的。

应用程序部分：

- 解析ELF程序
- 为应用程序创建独立的三级页表
- 映射应用程序跳板页
- 映射`trap`上下文
- 映射应用程序逻辑段
- 映射用户栈

任务创建部分：

- 任务控制块属性新增
- 修改`task_create`函数
- 设置每个应用程序的`trap`上下文：
  - `sepc`： 应用程序入口地址
  - 用户栈指针
  - 内核栈栈顶虚拟地址
  - `trap handler` 入口虚拟地址

1. 程序加载与映射

应用程序是以ELF格式组织的，我在这篇博客中：elf文件解析 | TimerのBlog (yanglianoo.github.io)对ELF文件的构成做了详细的解析，我们现在就需要编码把应用程序的数据解析出来。

首先在`loader.h`中定义ELF文件解析相关的数据结构：

c

```
//前16个字节
#define EI_NIDENT 16
//所有的ASCII码都可以用“\”加数字（一般是8进制数字）来
表示。
#define ELFMAG 0x464C457FU   // 0x464C457FU
"\177ELF"

#define EM_RISCV 0xF3  //risc-v 对应的 e_machine 的
value值

#define EI_CLASS     4 //EI_NIDENT 的 第四位
#define ELFCLASSNONE 0
#define ELFCLASS32   1
#define ELFCLASS64   2
#define ELFCLASSNUM  3

#define PT_LOAD 1

//ELF文件中段属性定义
#define PF_X 0x1
#define PF_W 0x2
#define PF_R 0x4


/**
 * @brief  elf header 结构体定义
 */
typedef struct {
    u8 e_ident[EI_NIDENT];
    u16 e_type;
    u16 e_machine;
    u32 e_version;
    u64 e_entry;
    u64 e_phoff;
    u64 e_shoff;
    u32 e_flags;
    u16 e_ehsize;
    u16 e_phentsize;
    u16 e_phnum;
    u16 e_shentsize;
    u16 e_shnum;
    u16 e_shstrndx;
} elf64_ehdr_t;

/**
 * @brief  program header 结构体定义
 */
typedef struct {
    u32 p_type;
    u32 p_flags;
    u64 p_offset;
    u64 p_vaddr;
    u64 p_paddr;
    u64 p_filesz;
    u64 p_memsz;
    u64 p_align;
} elf64_phdr_t;
```

然后再`loader.c`中新建一个`void load_app(size_t app_id)`的函数，这个函数做的事情就是上面提到的应用程序部分，我们直接先看代码：

c

```
static u8 flags_to_mmap_prot(u8 flags)
{
    return (flags & PF_R ? PTE_R : 0) |
           (flags & PF_W ? PTE_W : 0) |
           (flags & PF_X ? PTE_X : 0);
}

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;

    //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");
    }

    //记录APP程序的入口地址，为 main 函数地址
    u64 entry = (u64)ehdr->e_entry;
    //创建任务
    TaskControlBlock* proc = task_create_pt(app_id);
    //赋值任务的 entry
    proc->entry = entry;
    // Program Header 解析
    elf64_phdr_t *phdr;
    //遍历每一个逻辑段
    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 + metadata.start);
        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, metadata.start + 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);
```

```
            }


        }
    }

    // 映射应用程序用户栈开始地址
    proc->ustack =  2 * PAGE_SIZE + PGROUNDUP(proc->ustack);
    PhysPageNum ppn = kalloc();
    u64 paddr = phys_addr_from_phys_page_num(ppn).value;
    PageTable_map(&proc->pagetable,virt_addr_from_size_t(proc->ustack -
PAGE_SIZE),phys_addr_from_size_t(paddr), \
                  PAGE_SIZE, PTE_R | PTE_W | PTE_U);

}
```

1. 通过`get_app_data`拿到了应用程序的数据，这里传入的参数`app_id + 1`是因为`_num_app` 这个数组中第一位储存的是app的个数，所以比如`load_app(0)`，实际上取的就是`_num_app[1]`，这才对应上

2. 取出ELF文件的文件头，判断魔数，判断传入文件是否是 riscv64的。

3. 然后取出此应用程序的入口地址

4. 接下来就是创建任务：`task_create_pt`这个函数干的事其实就是：为应用程序创建独立的三级页表、映射应用程序跳板页、映射`trap`上下文，然后返回代表该任务的任务控制块指针，将此任务的入口地址设置为ELF文件中解析出来的。这里我们暂时先不看，后面来解析，只要知道干完了上面三件事即可

   c

   ```
   //创建任务
   TaskControlBlock* proc =
   task_create_pt(app_id);
   //赋值任务的 entry
   proc->entry = entry;
   ```

5. 接下来就是解析Program Header，遍历每一个逻辑段，如果此逻辑段是可被加载的，则将此逻辑段进行映射，同时将此逻辑段加载到物理内存中，我们先来看一下应用程序的逻辑段组成，如下图可以看见有两个逻辑段是需要被加载的，01 段的属性是 R E、大小0x1060超过了一页大小，虚拟地址是0x10000，02 段的属性是R W、大小是0x3eb8小于一页、起始虚拟地址是0x12000。

   对这两段进行映射时有几个需要注意的地方：

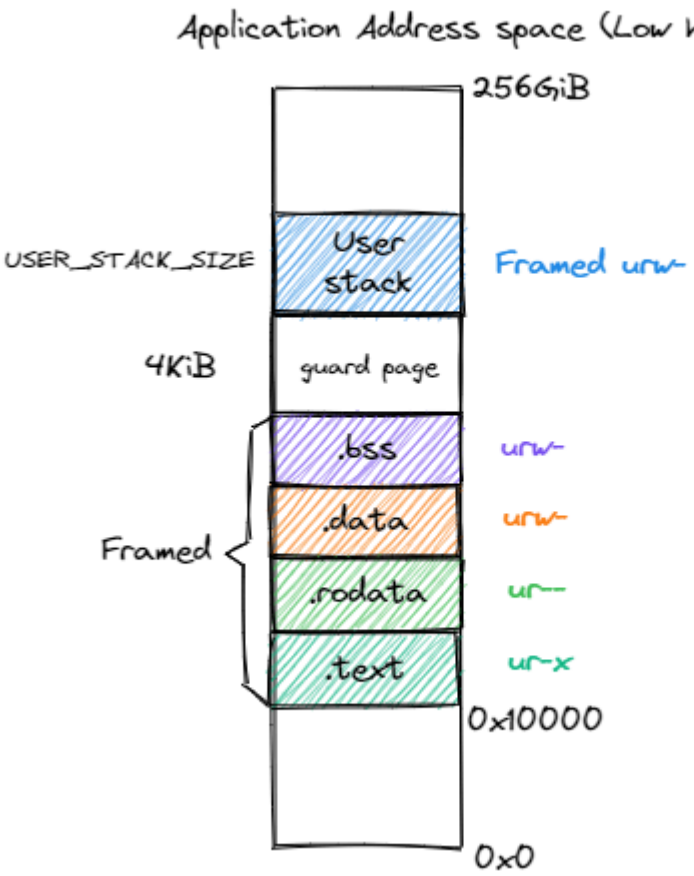
   - 对于大小超过一页的，需要向上对齐映射，就是要多映射一页，大小小于一页的就映射一页就对了；
   - 需要分配新的物理页来储存逻辑段的数据，分配内存是通过kalloc函数的，然后直接通过memcpy将数据拷贝到分配的物理内存处；
   - 在映射时ELF的可读可写可执行标志位和riscv的页表项不是相对应的，需要先转换一下，所以定义了一个flags_to_mmap_prot来用于转换标志位，同时这些段都是U模式下可访问的，因此映射的标志位需要或上PTE_U
   - 在调用PageTable_map函数开始映射时，映射的虚拟地址是start_va + j，就是因为如果这一段需要映射的内存大于一页，就需要一页一页内存

c

```
static u8 flags_to_mmap_prot(u8
flags)
{
    return (flags & PF_R ? PTE_R :
0) |
           (flags & PF_W ? PTE_W :
0) |
           (flags & PF_X ? PTE_X :
0);
}
```

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOPROC+0x3     0x000000000000207b 0x0000000000000000 0x0000000000000000
                 0x000000000000003e 0x0000000000000000  R      0x1
  LOAD           0x0000000000001000 0x0000000000010000 0x0000000000010000
                 0x0000000000001060 0x0000000000001060  R E    0x1000
  LOAD           0x0000000000000000 0x0000000000012000 0x0000000000012000
                 0x0000000000000000 0x00000000000003e8  RW     0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .riscv.attributes
   01     .text .rodata
   02     .bss
```

6. 在完成逻辑段映射后就需要映射用户程序的内核栈了,如下图内核栈栈顶的虚拟地址位于应用程序上面的两页，我通过proc->ustack来记录了应用程序结束的位置，然后proc->ustack = 2 * PAGE_SIZE + PGROUNDUP(proc->ustack);对这个结束位置向上对齐然后再加两页内存就得到了用户栈的栈顶的虚拟地址，然后进行映射，这里映射用户栈和映射内核栈同理，虚拟地址起始地址应该为guard page的顶部位置



至此通过load_app这个函数我们就完成了对应用程序数据的加载以及内存的映射。

2. 任务控制段修改

在上面load_app函数中，我们创建了一个TaskControlBlock* proc用来管理一个任务的具体信息，比如对任务的入口地址赋值，对任务的用户栈栈顶虚拟地址赋值：

c

```
//赋值任务的 entry
proc->entry = entry;
// 映射应用程序用户栈开始地址
proc->ustack =  2 * PAGE_SIZE + PGROUNDUP(proc-
>ustack);
```

相比于之前现在TaskControlBlock中多了一些信息：

C

```
typedef struct TaskControlBlock
{
    TaskState task_state;        //任务状态
    TaskContext task_context;    //任务上下文
    u64 trap_cx_ppn;             //Trap 上下文所在物
理地址
    u64  base_size;              //应用数据大小
    u64  kstack;                 //应用内核栈的虚拟
地址
    u64  ustack;                 //应用用户栈的虚拟
地址
    u64  entry;                  //应用程序入口地址
    PageTable pagetable;         //应用页表所在物理
页
}TaskControlBlock;
```

增加了：Trap 上下文所在物理地址 、应用数据大小、应用内核栈的虚拟地址、应用用户栈的虚拟地址、应用程序入口地址、应用页表所在物理页 。在加载程序以及进行映射时需要对这些属性赋值，这样我们就可以通过`TaskControlBlock`这个数据结构来管理和具体代表一个应用程序了。

其中`kstack`的赋值是在映射内核栈时搞定的：

```
/* 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 + PAGE_SIZE), phys_addr_from_size_t((u64)pa), \
                  PAGE_SIZE, PTE_R | PTE_W);
    // 给应用内核栈赋值
    p->kstack = va + 2 * PAGE_SIZE;
  }
}
```

3. 创建页表映射跳板页和trap上下文

在`load_app`函数中，我们调用了一个名为`task_create_pt`的函数，这个函数会完成应用程序页表的建立，同时映射跳板页和trap上下文页，然后返回一个代表一个应用程序的任务控制块，这个函数定义在`task.c`中

C

```
TaskControlBlock* task_create_pt(size_t
app_id)
{
  if(_top < MAX_TASKS)
  {

    //为应用程序分配一页内存用与存放trap
    proc_trap(&tasks[app_id]);
    //为用户程序创建页表，映射跳板页和trap上
下文页
    proc_pagetable(&tasks[app_id]);
    _top++;
  }

  return &tasks[app_id];
}
```

此函数传入的参数为app_id，然后争对此应用程序首先分配了一页内存用于存放trap页，然后为用户程序创建页表，映射跳板页和trap上下文页。

C

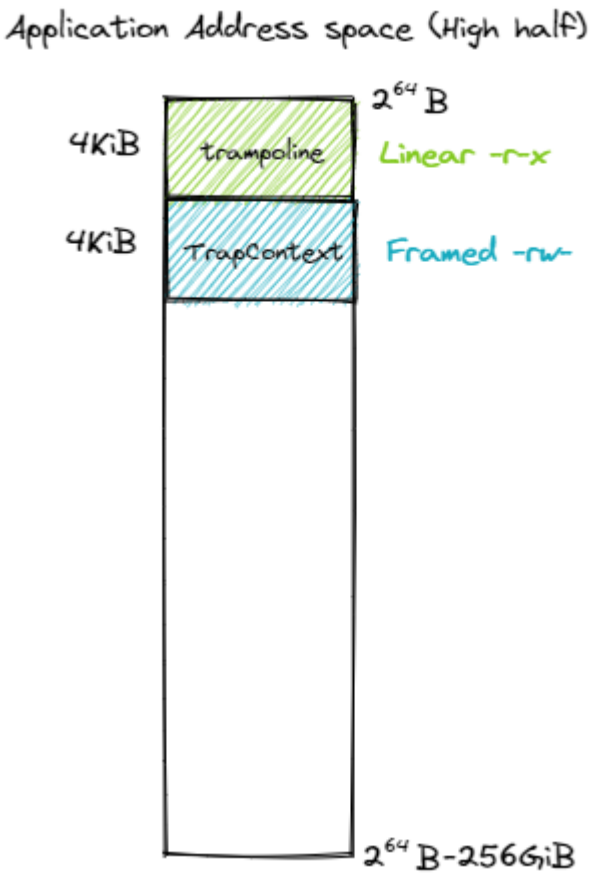
```
/* 为每个应用程序分配一页内存用与存放trap，同时初始化任务上
下文 */
void proc_trap(struct TaskControlBlock *p)
{
  // 为每个程序分配一页trap物理内存
  p->trap_cx_ppn =
phys_addr_from_phys_page_num(kalloc()).value;
  printk("trap value : %p\n",p->trap_cx_ppn);
  // 初始化任务上下文全部为0
  memset(&p->task_context, 0 ,sizeof(p->task_context));
}
```

每个应用程序都需要一个trap页来存储自己的trap上下文，因此需要事先分配一页内存，然后对p->trap_cx_ppn赋值，分配完毕后将任务上下文全部清零。

C

```
extern char trampoline[];
/* 为用户程序创建页表，映射跳板页和trap上下文页*/
void proc_pagetable(struct TaskControlBlock *p)
{
  // 创建一个空的用户的页表，分配一页内存
  PageTable pagetable;
  pagetable.root_ppn = kalloc();

  //映射跳板页

PageTable_map(&pagetable,virt_addr_from_size_t(TRAMPOLINE),phys_addr_from_size_t((u64)trampol
ine),\
                PAGE_SIZE , PTE_R | PTE_X);
  printk("finish user TRAMPOLINE map!\n");
  //映射用户程序的trap页
  PageTable_map(&pagetable,virt_addr_from_size_t(TRAPFRAME),phys_addr_from_size_t(p-
>trap_cx_ppn), \
                PAGE_SIZE, PTE_R | PTE_W );
  printk("finish user TRAPFRAME map!\n");
  p->pagetable = pagetable;
  printk("p->pagetable:%p\n",p->pagetable.root_ppn.value);
}
```

然后就是创建页表，和之前对内核地址映射的操作同理，分配一页内存来存放根页表，然后依次映射跳板页和trap页，这里需要注意的是在上一篇博客中提到所有的应用程序都是共享同一个跳板页的，所以是映射到同一个物理地址，然后呢各自有自己的trap页，因此需要分配一个物理内存来映射。跳板页的虚拟地址位于应用地址空间最顶端的地址，trap页位于其下一页。



映射完成后对`p->pagetable`赋值。

4. 改进Trap处理的实现

c

```
void trap_from_kernel()
{
        panic("a trap from kernel!\n");
}

void set_kernel_trap_entry()
{
        w_stvec((reg_t)trap_from_kernel);
}

void set_user_trap_entry()
{
        w_stvec((reg_t)TRAMPOLINE);
}
void trap_handler()
{
        set_kernel_trap_entry();
        TrapContext* cx = get_current_trap_cx();

    reg_t scause = r_scause();
        reg_t cause_code = scause & 0xfff;
        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->a0 = __SYSCALL(cx->a7,cx->a0,cx->a1,cx-
>a2);
                        cx->sepc += 8;
                        break;
                default:
                        printk("undfined exception
scause:%x\n",scause);
                        break;
                }
        }

        trap_return();
}
```

由于应用的 Trap 上下文不在内核地址空间，因此我们调用 current_trap_cx 来获取当前应用的 Trap 上下文，这个函数定义在task.c中：

c

```
/* 返回当前执行的应用程序的trap上下文的
地址 */
u64 get_current_trap_cx()
{
  return tasks[_current].trap_cx_ppn;
}
```

注意到，在 trap_handler 的开头还调用 set_kernel_trap_entry 将 stvec 修改为同模块下另一个函数 trap_from_kernel 的地址。这就是说，一旦进入内核后再次触发到 S态 Trap，则硬件在设置一些 CSR 寄存器之后，会跳过对通用寄存器的保存过程，直接跳转到 trap_from_kernel 函数，在这里直接 panic 退出。这是因为内核和应用的地址空间分离之后，U态 –> S态 与 S态 –> S态 的 Trap 上下文保存与恢复实现方式/Trap 处理逻辑有很大差别。这里为了简单起见，弱化了 S态 –> S态的 Trap 处理过程：直接 panic

> 在这里我之前有个误区：在改进 Trap 处理的实现时，通过set_kernel_trap_entry函数将S态的异常处理地址设置成了trap_from_kernel函数，那时钟中断也会进入这个函数来处理吧，按照之前分时多任务的处理逻辑，当检测到是时钟中断时会进入_alltraps函数然后跳转到trap_handler来分发，从而进行调度。而在这里修改后，调度的过程是怎么发生的，我没想明白。
>
> 这个问题下面来解释

在 trap_handler 完成 Trap 处理之后，我们需要调用 trap_return 返回用户态：

c

```
void trap_return()
{
    /* 把 stvec 设置为内核和应用地址空间共享的跳板页面的起始地址 */
    set_user_trap_entry();
    /* Trap 上下文在应用地址空间中的虚拟地址 */
    u64 trap_cx_ptr = TRAPFRAME;
    /* 要继续执行的应用地址空间的 token */
    u64  user_satp = current_user_token();

    u64  restore_va = (u64)__restore - (u64)__alltraps + TRAMPOLINE;

        asm volatile (
                        "fence.i\n\t"
                        "mv a0, %0\n\t"  // 将trap_cx_ptr传递给a0寄存
器
                        "mv a1, %1\n\t"  // 将user_satp传递给a1寄存器
                        "jr %2\n\t"      // 跳转到restore_va的位置执行
代码
                        :
                        : "r" (trap_cx_ptr),
                        "r" (user_satp),
                        "r" (restore_va)
                        : "a0", "a1"
                );

}
```

- 在 trap_return 的开始处就调用 set_user_trap_entry ，来让应用 Trap 到 S 的时候可以跳转到 __alltraps 。注：我们把 stvec 设置为内核和应用地址空间共享的跳板页面的起始地址TRAMPOLINE 而不是编译器在链接时看到的 __alltraps 的地址。这是因为启用分页模式之后，内核只能通过跳板页面上的虚拟地址来实际取得 __alltraps 和 __restore 的汇编代码。

  > 这里就能回答我上面那个问题，当应用程序在执行时，此时产生了时钟中断，由于此时stvec寄存器的值我们在 trap_return 时设置成了TRAMPOLINE，所以此时会进入__alltraps 函数执行，然后跳转到trap_handler进行处理来调用调度函数进行函数切换

- 准备好 __restore 需要两个参数：分别是 Trap 上下文在应用地址空间中的虚拟地址和要继续执行的应用地址空间的 token 。最后我们需要跳转到 __restore ，以执行：切换到应用地址空间、从 Trap 上下文中恢复通用寄存器、 sret 继续执行应用。它的关键在于如何找到 __restore 在内核/应用地址空间中共同的虚拟地址。

- 由于 __alltraps 是对齐到地址空间跳板页面的起始地址 TRAMPOLINE 上的， 则 __restore 的虚拟地址只需在 TRAMPOLINE 基础上加上 __restore 相对于 __alltraps 的偏移量即可。这里 __alltraps 和 __restore 都是指编译器在链接时看到的内核内存布局中的地址。

- 使用 `fence.i` 指令清空指令缓存 i-cache 。这是因为，在内核中进行的一些操作可能导致一些原先存放某个应用代码的物理页帧如今用来存放数据或者是其他应用的代码，i-cache 中可能还保存着该物理页帧的错误快照。因此我们直接将整个 i-cache 清空避免错误。接着使用 `jr` 指令完成了跳转到 `__restore` 的任务。

5. 初始化任务

完成上面的步骤后，接下来就万事具备只欠东风了，回想之前我们要从内核态跳转到用户态执行应用程序之前需要填充此应用程序的`trap`上下文和任务上下文，之前是在`task_create`函数中完成的，现在对其就行了修改：

c

```
extern u64 kernel_satp;
void app_init(size_t app_id)
{
    TrapContext* cx_ptr = tasks[app_id].trap_cx_ppn;
    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);
    // 设置程序入口地址
    cx_ptr->sepc = tasks[app_id].entry;
    printk("cx_ptr->sepc:%p\n",cx_ptr->sepc);
    //
    cx_ptr->sstatus = sstatus;
    // 设置用户栈虚拟地址
    cx_ptr->sp = tasks[app_id].ustack;
    printk("cx_ptr->sp:%p\n",cx_ptr->sp);
    // 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 = tasks[app_id].kstack;
    printk("cx_ptr->kernel_sp:%p\n",cx_ptr->kernel_sp);
    // 设置内核trap_handler的地址
    cx_ptr->trap_handler = (u64)trap_handler;
    printk("cx_ptr->trap_handler:%p\n",cx_ptr->trap_handler);

    /* 构造每个任务任务控制块中的任务上下文，设置 ra 寄存器为 trap_return 的入口
地址*/
    tasks[app_id].task_context = tcx_init((reg_t)cx_ptr);
    // 初始化 TaskStatus 字段为 Ready
    tasks[app_id].task_state = Ready;
}
```

当每个应用第一次获得 CPU 使用权即将进入用户态执行的时候，它的内核栈顶放置着我们在内核加载应用的时候 构造的一个任务上下文;在 `__switch` 切换到该应用的任务上下文的时候，内核将会跳转到 `trap_return` 并返回用户态开始该应用的启动执行。

在开启地址空间后，无论是从内核切换到应用程序还是从应用程序切换到内核都需要对`satp`的值进行切换，因此需要在任务上下文中保存内核的`satp`的值，在内核中需要知道当前执行的应用程序的`satp`的值。

综上所述，我们需要在应用trap上下文中保存：程序入口地址、用户栈虚拟地址、内核页表token、内核栈虚拟地址、内核`trap_handler`的地址。需要在任务上下文中：将trap上下文的地址放到任务上下文的`sp`寄存器中，将任务上下文的返回地址设置为`trap_return`

```
struct TaskContext tcx_init(reg_t kstack_ptr) {
    struct TaskContext task_ctx;

    task_ctx.ra = trap_return;
    task_ctx.sp = kstack_ptr;
    task_ctx.s0 = 0;
    task_ctx.s1 = 0;
    task_ctx.s2 = 0;
    task_ctx.s3 = 0;
    task_ctx.s4 = 0;
    task_ctx.s5 = 0;
    task_ctx.s6 = 0;
    task_ctx.s7 = 0;
    task_ctx.s8 = 0;
    task_ctx.s9 = 0;
    task_ctx.s10 = 0;
    task_ctx.s11 = 0;

    return task_ctx;
}
```

6. 改进sys_write

这里为啥需要对`sys_write`进行改写呢，那是因为假设一个应用程序在应用地址空间调用了`sys_write`函数，其中有个参数是：`char * buf`，这里代表了字符串储存的地址，但是这是应用地址空间的地址，进入内核态后切换到内核地址空间`char * buf`所代表的字符串的地址我们不能直接在内核地址空间下访问，需要转换成实际的物理地址去访问，需要我们手动查页表去访问，因此在`sys_call.c`中定义了一个辅助函数：

c

```c
void translated_byte_buffer(const char* data , size_t len)
{
    //拿到当前应用程序的 satp
    u64  user_satp = current_user_token();
    PageTable  pt ;
    //根据 satp 找到应用的根页表
    pt.root_ppn.value = MAKE_PAGETABLE(user_satp);
        //定义字符串起始地址和结束地址
    u64 start_va = data;
    u64 end_va = start_va + len;
    VirtPageNum vpn =
floor_virts(virt_addr_from_size_t(start_va));
    //根据虚拟页号查找页表
    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;
        //打印字符串
    u64 data_d = phyaddr + page_offset;
    char *data_p = (char*) data_d;
    printk("%s",data_p);

}
```

通过这个函数就可以拿到实际的物理页上的字符串的数据，并将其打印出来

```c
void __sys_write(size_t fd, const char* data, size_t len)
{

    if(fd == stdout || fd == stderr)
    {
        translated_byte_buffer(data,len);
    }
    else
    {
        panic("Unsupported fd in sys_write!");
    }
}
```

7. 测试

修改应用程序：

time.c

c

```
#include <timeros/types.h>
#include <timeros/syscall.h>
#include <timeros/string.h>
int main()
{
    uint64_t current_timer = 0;
    while (1)
    {
        current_timer = sys_gettime();

printf("current_timer:%x\n",current_timer);
    }
    return 0;
}
```

write.c

c

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

    const char *message = "task write is
running!\n";
    while (1)
    {
       printf(message);
    }
    return 0;
}
```

Makefile

makefile

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

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

LIB = ../lib

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 $^

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

上面这两个程序都会用到lib目录下的函数，因此将lib目录下的源文件页加入编译，同时我把app.c修改了一下放到了lib目录下,write.c和time.c都是调用了app.c的函数来执行系统调用

c

```c
#include <timeros/os.h>
uint64_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t
arg3) {

    register uintptr_t a0 asm ("a0") = (uintptr_t)(arg1);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg2);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg3);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(id);

    asm volatile ("ecall"
                  : "+r" (a0)
                  : "r" (a1), "r" (a2), "r" (a7)
                  : "memory");
    return a0;
}

uint64_t sys_write(size_t fd, const char* buf, size_t len)
{
    return syscall(__NR_write,fd,buf, len);
}

uint64_t sys_yield()
{
    return syscall(__NR_sched_yield,0,0,0);
}

uint64_t sys_gettime()
{
    return syscall(__NR_gettimeofday,0,0,0);
}
```

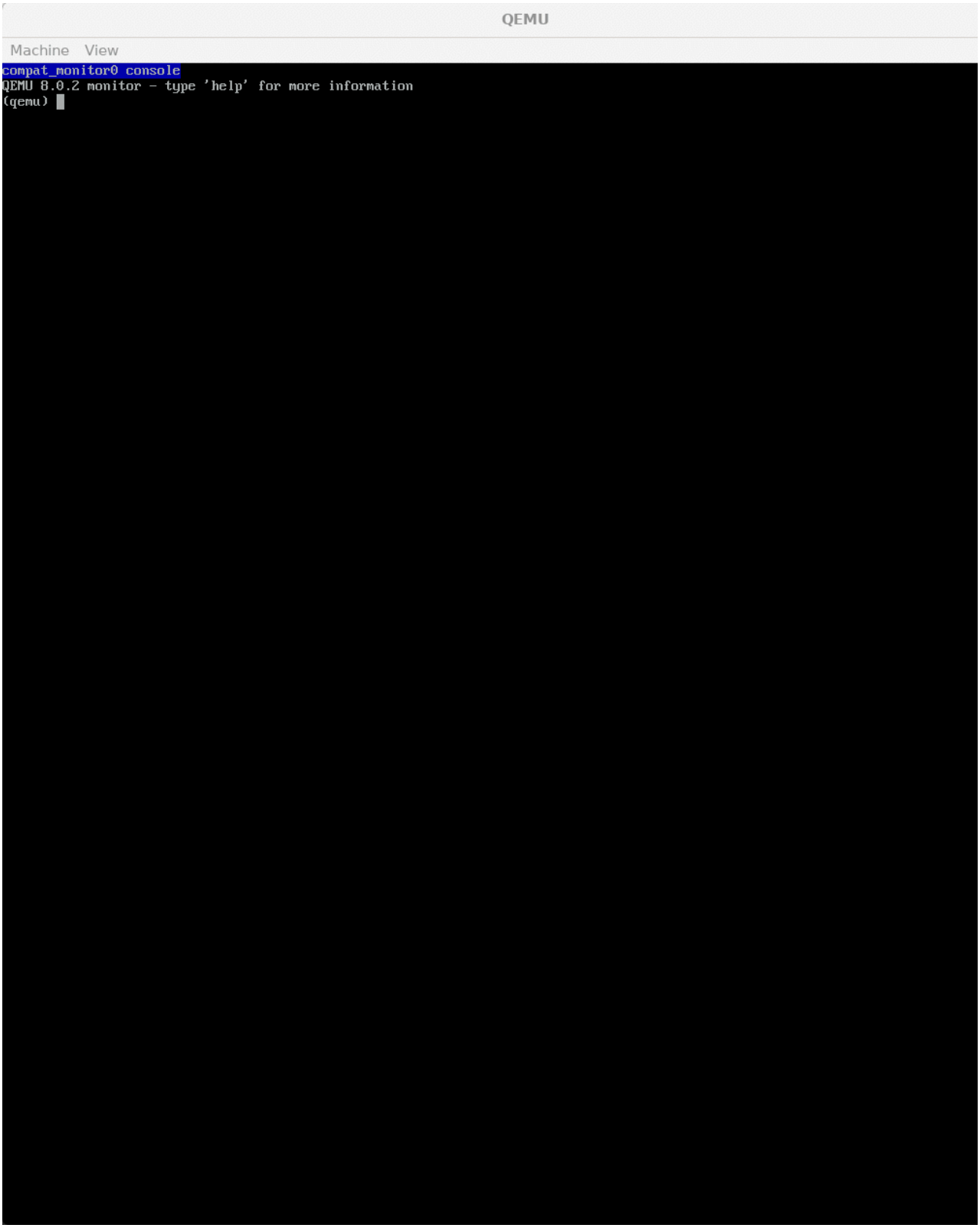
先编译应用程序：

```
timer@DESKTOP-JI9EVEH:~/quard-star/os/user$ make time
riscv64-unknown-elf-gcc -nostdlib -fno-builtin -mcmodel=medany -I../include -T user.ld -Wl,-Map=time.map -o bin/time tim
e.c ../lib/printf.c ../lib/string.c ../lib/app.c ../lib/vsprintf.c
../lib/app.c: In function 'sys_write':
../lib/app.c:18:34: warning: passing argument 3 of 'syscall' makes integer from pointer without a cast [-Wint-conversio
]
   18 |     return syscall(__NR_write,fd,buf, len);
      |                                  ^~~
      |                                  |
      |                                  const char *
../lib/app.c:2:47: note: expected 'reg_t' {aka 'long long unsigned int'} but argument is of type 'const char *'
    2 | uint64_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t arg3) {
      |                                         ~~~~~~~^~~~
timer@DESKTOP-JI9EVEH:~/quard-star/os/user$ make write
riscv64-unknown-elf-gcc -nostdlib -fno-builtin -mcmodel=medany -I../include -T user.ld -Wl,-Map=write.map -o bin/write w
rite.c ../lib/printf.c ../lib/string.c ../lib/app.c ../lib/vsprintf.c
../lib/app.c: In function 'sys_write':
../lib/app.c:18:34: warning: passing argument 3 of 'syscall' makes integer from pointer without a cast [-Wint-conversio
]
   18 |     return syscall(__NR_write,fd,buf, len);
      |                                  ^~~
      |                                  |
      |                                  const char *
../lib/app.c:2:47: note: expected 'reg_t' {aka 'long long unsigned int'} but argument is of type 'const char *'
    2 | uint64_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t arg3) {
      |                                         ~~~~~~~^~~~
```

修改main函数：将两个程序加载和初始化，设置内核的trap的stvec的地址，然后开启时钟，开始执行

```
void os_main()
{
    printk("hello timer os!\n");

    // 内存分配器初始化
    frame_alloctor_init();

    //初始化内存
    kvminit();

    load_app(0);
    app_init(0);
    load_app(1);
    app_init(1);
    //映射内核
    kvminithart();

    //trap初始化
    set_kernel_trap_entry();

    timer_init();

    run_first_task();

}
```

编译内核和执行：

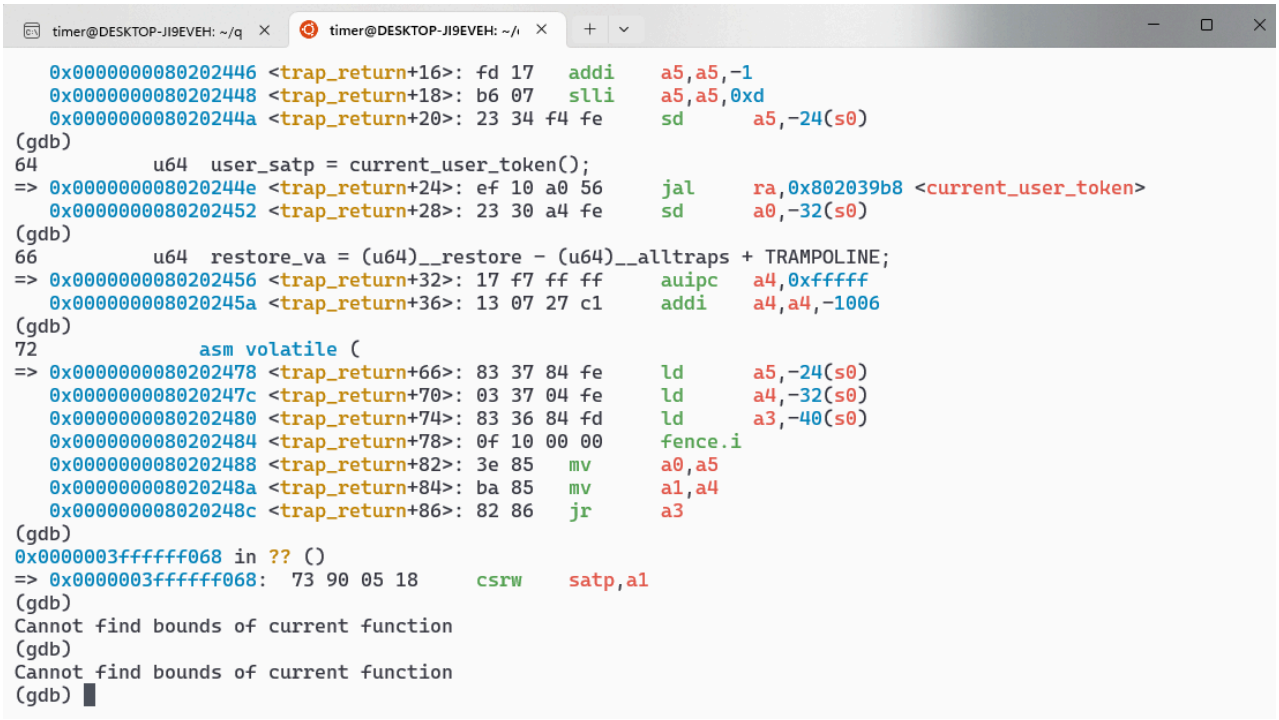


OK，验证成功。

参考链接

> 基于地址空间的分时多任务 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)

最后说一下这一节的代码涉及到很多细节，需要很耐心的调试，我是用GDB用si指令一步一步跟进汇编然后看地址，看寄存器的值来调试最后才跑通的，可以看见在生成应用程序时我是用objdump生成了汇编代码，这也是我当时调试的产物，这里说几个调试相关的问题：



在上图中我用n指令去调试，当函数从trap_return跳转到_restore函数时，就会出现`Cannot find bounds of current function`的问题，此时就得用`si`单步汇编调试了。

第二个是好像现在开启中断后我是没法调试的，不知道咋解决……

相关推荐