LAB5 实验报告

ELF文件格式

ELF文件是一种用于二进制文件、可执行文件、目标代码、共享库和core转存格式文件。是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是Linux的主要可执行文件格式。

1. **ELF Header（ELF文件头）**：
   - 这是文件的开始部分，包含了用于解释文件本身的元数据。
   - 重要字段包括：魔数（Magic Number）、文件类别（32位或64位）、字节序（大端或小端）、文件版本、入口点地址、程序头表位置、节头表位置等。

```
/* 文件头 */
struct elfhdr {
    uint32_t e_magic;     // 必须等于ELF_MAGIC（ELF魔数）
    uint8_t e_elf[12];    // ELF标识数组
    uint16_t e_type;      // 文件类型：1=relocatable（可重定位文件），
2=executable（可执行文件），3=shared object（共享对象文件），4=core image（核心映像文件）
    uint16_t e_machine;   // 机器类型：3=x86，4=68K等
    uint32_t e_version;   // 文件版本，总是1
    uint64_t e_entry;     // 如果是可执行文件，这是程序入口点（Entry point）
    uint64_t e_phoff;     // 程序头部偏移量（Program header table file
offset），如果没有程序头部则为0
    uint64_t e_shoff;     // 节头部偏移量（Section header table file offset），
如果没有节头部则为0
    uint32_t e_flags;     // 特定于架构的标志（Architecture-specific flags），通
常为0
    uint16_t e_ehsize;    // ELF头部的大小（ELF header size）
    uint16_t e_phentsize; // 程序头部表项大小（Program header table entry size）
    uint16_t e_phnum;     // 程序头部表项数量（Number of entries in the program
header table），如果没有程序头部则为0
    uint16_t e_shentsize; // 节头部表项大小（Section header table entry size）
    uint16_t e_shnum;     // 节头部表项数量（Number of entries in the section
header table），如果没有节头部则为0
    uint16_t e_shstrndx;  // 包含节名称字符串的节头部索引（Section header table
index of the entry associated with the section name string table）
};
```

2. **Program Header Table（程序头表）**：
   - 紧跟在ELF头后，这个表描述了如何将ELF文件的不同部分映射到进程的地址空间。
   - 每个表项提供了一个段的信息，包括类型（比如是否需要加载到内存）、文件内偏移、在内存中的虚拟地址、在内存和文件中的大小、以及所需的内存和文件对齐方式。

```c
/* 程序头表(Program header table） */
struct proghdr {
    uint32_t p_type;   // 段类型：可加载代码或数据，动态链接信息等。
    uint32_t p_flags;  // 段的读/写/执行权限标志
    uint64_t p_offset; // 文件中的段偏移量
    uint64_t p_va;     // 虚拟地址，用于映射段
    uint64_t p_pa;     // 物理地址，未使用
    uint64_t p_filesz; // 文件中的段大小
    uint64_t p_memsz;  // 内存中的段大小（如果包含bss则更大）
    uint64_t p_align;  // 所需对齐方式，通常是硬件页面大小
};
```

3. **Section Header Table（节头表）**：
   - 文件的最后部分通常是节头表，它描述了文件中所有的节（Section）和它们的属性。
   - 这个表主要用于链接视图，而程序头表用于执行视图。节可以包含代码、数据、符号表、重定位信息等。
4. **Text Section（文本段）**：
   - 这通常是代码段，包含了程序的机器码。
5. **Data Section（数据段）**：
   - 包含了初始化的全局变量和静态变量。
6. **BSS Section（BSS段）**：
   - 包含了未初始化的全局变量和静态变量。在文件中不占用空间，但在内存中需要被清零。

背景知识补充

1. **CPU模式（特权级）**：
   - CPU有不同的运行模式，以限制某些敏感操作的执行。最常见的模式是用户模式（User Mode, 通常称为U mode）和内核模式（Kernel Mode, 在RISC-V中通常称为S mode或Supervisor Mode）。
   - 用户模式用于运行普通应用程序。这些程序受限于它们能执行的操作，以保护计算机系统不受恶意软件的伤害。
   - 内核模式用于运行操作系统代码，它允许执行所有CPU指令和访问所有硬件资源。
2. **系统调用（Syscalls）**：
   - 系统调用是程序在用户模式下请求操作系统执行操作的方式。这些操作包括文件操作、进程控制等，这些通常不能由用户模式下的程序直接执行。
3. **中断和异常**:
   - 中断是指CPU的正常执行流程被打断的事件，通常是由外部设备触发的。
   - 异常是由程序执行流程中发生的错误或特殊情况引起的中断，例如除以零或尝试执行非法指令。
4. `ecall` **和** `ebreak` **指令**:
   - `ecall`（Environment Call）是用于从用户模式切换到内核模式的指令，触发系统调用。
   - `ebreak`（Environment Break）是用于触发断点异常的指令，常用于调试。
5. **寄存器和状态**:

- CPU中有许多小的存储单元，称为寄存器，用于存储指令、数据和状态信息。
- `sstatus` 是一个寄存器，存储了当前CPU的各种状态，比如当前是在用户模式还是内核模式运行。

6. **SPP（Previous Privilege Mode）**：
   - `sstatus` 寄存器的 `SPP` 字段存储了进入异常或中断前的CPU模式。
   - 在RISC-V架构中，`SPP` 位为0表示用户模式，为1表示内核模式。

练习1: 加载应用程序并执行

- **load_icode 函数执行流程**：给用户进程建立一个能让它正常运行的用户环境
  - 调用mm_create函数来申请进程的内存管理数据结构mm所需内存空间，并对mm进行初始化
  - 调用setup_pgdir来申请一个页目录表所需的内存空间，内核页表（boot_pgdir所指）的内容拷贝到此新目录表中，最后mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核
  - 将二进制的TEXT/DATA段复制到进程的内存空间，并构建BSS部分，根据应用程序执行码的起始位置来解析此ELF格式的执行程序，调用mm_map函数设置新vma，调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中
  - 需要给用户进程设置用户栈，为此调用mm_mmap函数建立用户栈的vma结构
  - 至此,进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好
  - 重新设置进程的 trapframe，使得执行中断返回指令 iret 后，CPU 会切换回用户态，跳转到用户进程的第一条指令执行，并确保在用户态可以响应中断。

```
    tf->gpr.sp=USTACKTOP;//需要将esp设置为用户栈的栈顶，直接使用之前建立用户栈时的参数
USTACKTOP就可以
    tf->epc=elf->e_entry;
    tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

其中：需要将esp设置为用户栈的栈顶，直接使用之前建立用户栈时的参数USTACKTOP就可以，ecp指向elf文件加载到内存之后的入口。

- **用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过**:
  - 1.用户进程被选择占用，执行宏"KERNEL_EXECVE(hello)"，调用kernel_execve（），触发一个特殊的异常，从而将相应参数传给syscall（）函数

```
        case CAUSE_BREAKPOINT:
            cprintf("Breakpoint\n");
            if(tf->gpr.a7 == 10){
                tf->epc += 4;//将异常程序计数器（epc）的值增加4，因为RISC-V中的大部分指令都是4个字节的
                //sepc寄存器是产生异常的指令的位置，在异常处理结束后，会回到sepc的位置继续执行
                //对于ecall，我们希望sepc寄存器要指向产生异常的指令(ecall)的下一条指令
                //否则就会回到ecall执行再执行一次ecall，无限循环
                syscall();// 进行系统调用处理
                kernel_execve_ret(tf,current->kstack+KSTACKSIZE);
            }
            break;
```

- syscall()会把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理,最终调用 `do_execve` 函数完成应用程序的加载
- 在`do_execve`中，调用 exit_mmap(mm) 和 put_pgdir(mm) 来回收当前进程的内存空间，后调用load_icode去加载ELF二进制格式文件到内存
- 此时返回到trap.c中，执行`kernel_execve_ret(tf,current->kstack+KSTACKSIZE)`,从而切换到用户态，跳转到程序的入口

练习2: 父进程复制自己的内存空间给子进程

在`do_fork()`中，调用`copy_mm(clone_flags,proc)`,其中调用`dup_mmap(mm, oldmm)`,其中调用`copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end, share)`,完成内存区域的复制。其遍历parent指定的某段内存空间里的每个虚拟页，如果虚拟页存在，就为`child`的同一个虚拟地址申请分配一个物理页,将前者的所有内容复制给后者，最后为 child 的这个虚拟地址和物理页建立映射关系

- **`copy_range`实现父进程到子进程内容的拷贝**

```
            void *src_kvaddr = page2kva(page); // 得到源页的内核虚拟地址
            void *dst_kvaddr = page2kva(npage); // 得到目标新页的内核虚拟地址
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE); // 复制内存
            ret = page_insert(to, npage, start, perm); // 建立新页的映射
```

- **实现`Copy on Write`机制:**
  - Copy-on-Write（COW）机制使得进程在执行fork系统调用时，不立即复制父进程的内存内容到子进程，而是让父子进程暂时共享同一物理内存页。仅当任一进程尝试修改内存时，系统才会为该进程创建一个私有的物理内存页，并将共享内容复制到这个新页中，以便进程在其私有内存页上进行修改。
  - 在fork操作过程中，应避免直接复制内存。具体地，在如copy_range函数的内存复制部分，应将子进程的虚拟页映射到与父进程相同的物理页面，并将这两个进程的虚拟页对应的页表项(PTE)设置为不可写，同时标记为共享页。这样，任何对共享页的写操作都会触发页访问异常(page fault)，将控制权交给操作系统。

- 对于page fault处理，应在中断服务例程(ISR)中识别引发异常的操作是否是对共享页的写尝试。如果是，则系统会分配一个新的物理页，将共享页的内容复制到新页，并更新引发异常的进程的页表，将新物理页与出错的线性地址映射，并设置PTE为非共享。同时，系统还需检查原共享物理页是否仍被其他进程共享，如果不再共享，则对应的虚拟地址的PTE会被更新，移除共享标记，恢复写权限。这样，page fault处理完成后，进程就能在其私有内存页上正常执行写操作。

练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

- **调用过程**
  - fork(用户态 ulib.c)--->sys_fork(用户态 syscall.c)---> syscall(SYS_exit, error_code)(用户态)--->接受参数并存在 trapframe 中后调用 `ecall` 产生一个中断，进入内核态的异常处理--->exception_handler()(内核态)--->调用内核态的 `syscall` 函数--->sys_fork()--->do_fork(0, stack, tf)--->服务完成后，调用 iret 返回用户态kernel_execve_ret()
- **fork**
  - **do_fork工作**：创建新用户进程
    - alloc_proc：分配并初始化PCB
    - 调用setup_stack()函数为进程分配一个内核栈
    - copy_mm：根据 clone_flag 标志复制或共享进程内存管理结构
    - copy_thread(proc,stack,tf)：复制父进程的中断帧和上下文信息
    - `get_pid` 给进程分配一个 pid，`hash_proc` 和 `set_links` 把 PCB 放入 hash_list 和 proc_list 两个全局进程链表中，设置 process
    - wakeup_proc(proc)：唤醒新进程 proc->state = PROC_RUNNABLE
- **exec**
  - **do_execve工作**：回收当前进程的内存空间，调用了load_icode去加载ELF二进制格式文件到内存并执行
    - user_mem_check：检查name的内存空间能否被访问
    - 检查是否有进程需要此进程占用的内存空间，如果没有，就把当前占用的内存清空（包括进程页表本身），然后重新分配内存
    - `load_icode`：把应用程序执行码加载到用户虚拟空间中
- **wait**
  - **do_wait工作**：父进程等待子进程exit,并完成子进程的回收
    - 找一个进程 id 号为 pid 的退出状态的子进程，如果pid==0，则随意找一个处于退出状态的子进程
    - 如果没找到，则父进程重新进入睡眠状态，之后调用schedule函数挂起自己，选择其他进程执行。
    - 如果找到了，将其 PCB 从进程链表 proc_list 和 hash_list 中删除，释放它的内核栈和 PCB，child 结束它的执行过程，所有资源都被释放。

- **exit**
  - **do_exit工作**：执行一系列的清理操作以确保进程资源被正确地释放，并且通知进程的父进程来处理退出的子进程
    - 判断该进程是否是用户进程，如果是，就回收它的用户态虚拟内存空间
    - 设置进程状态为PROC_ZOMBIE,并设置相应的错误码，表示这个进程已经不再被调度
    - 遍历当前进程的所有子进程，如果当前进程还有子进程（孤儿进程），则需要把这些子进程的父进程指针设置为内核线程initproc，且各个子进程指针需要插入到initproc的子进程链表中。如果某个子进程的执行状态是PROC_ZOMBIE，则需要唤醒initproc来完成对此子进程的最后回收工作。
- **执行状态生命周期**