LAB5

2112547张玉硕 2113099祝天智 2111288杜金轩

练习1: 加载应用程序并执行（需要编码）

加载的过程

在`init_main`里 新建了一个内核进程，执行函数`user_main()`,这个内核进程里我们将要开始执行用户进程

```
int pid = kernel_thread(user_main, NULL, 0);
```

在`user_main()`所做的，就是执行了

```
kern_execve("exit",
_binary_obj___user_exit_out_start,_binary_obj___user_exit_out_size)
```

就是加载了存储在这个位置的程序`exit`并在`user_main`这个进程里开始执行。这时`user_main`就从内核进程变成了用户进程。

```
static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
 //   ret = do_execve(name, len, binary, size);
    asm volatile(
        "li a0, %1\n"
        "lw a1, %2\n"
        "lw a2, %3\n"
        "lw a3, %4\n"
        "lw a4, %5\n"
        "li a7, 10\n"
        "ebreak\n"
        "sw a0, %0\n"
        : "=m"(ret)
        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
        : "memory");
    cprintf("ret = %d\n", ret);
    return ret;
}
```

`do_execve()`里面只是构建了用户程序运行的上下文，但是并没有完成切换。上下文切换实际上要借助中断处理的返回来完成。直接调用`do_execve()`是无法完成上下文切换的。如果是在用户态调用`exec()`，系统调用的`ecall`产生的中断返回时， 就可以完成上下文切换。

由于目前我们在S mode下，所以不能通过`ecall`来产生中断。我们这里采取一个取巧的办法，用`ebreak`产生断点中断进行处理，通过设置`a7`寄存器的值为10说明这不是一个普通的断点中断，而是要转发到`syscall()`,实现了在内核态使用系统调用。

do_execv函数调用`load_icode`（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序。你需要补充`load_icode`的第6步，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好`proc_struct`结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

```
    // Set user stack top// tf->gpr.sp should be user stack top (the value of sp)
    tf->gpr.sp = USTACKTOP;// 栈
    // Set entry point of user program// tf->epc should be entry point of user program (the value of sepc)
    tf->epc = elf->e_entry;// 用户程序入口
    // Set status register appropriately// tf->status should be appropriate for user program (the value of sstatus)
    tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;// 状态寄存器
```

这段代码的目的是在用户态执行程序时，正确地设置陷阱帧中的相关信息，包括用户栈指针、用户程序入口地址以及状态寄存器的值，以确保在中断或异常发生时，可以正确地保存和恢复用户程序的状态。

- 请简要描述这个用户态进程被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。

```
int
do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) {
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) {
        len = PROC_NAME_LEN;
    }

    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);

    if (mm != NULL) {
        cputs("mm != NULL");
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
```

```
        current->mm = NULL;
    }
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

保留pid但是要把原来的内存空间给清掉`lcr3(boot_cr3);`页表指向内核页表

```
 if (mm_count_dec(mm) == 0) {
           exit_mmap(mm);  // 释放当前进程的内存映射
           put_pgdir(mm);  // 释放页目录
           mm_destroy(mm);  // 销毁内存管理结构
       }
   current->mm = NULL;  // 将当前进程的内存管理结构指针置为空
```

```
// 通过load_icode载入程序（二进制代码）到内存中
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;  // 如果载入失败，跳转到 execve_exit 标签处进行处理
    }
```

```
    //(1) 创建新的内存管理空间
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
```

```
    //(2)创建新的页表
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
```

```
    //(3) 加载执行代码内容：代码段数据段的位置
    struct Page *page;
    struct elfhdr *elf = (struct elfhdr *)binary;// 文件头
    struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff);// 根据文件头偏移
```

```
    //(3.5)设置vma（管理合法的地址空间）设置权限
        vm_flags = 0, perm = PTE_U | PTE_V;// 标志，根据文件头设置对应vma的权限
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;// 可执行
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;// 可写
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;// 可读

        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {// 设置新的vma
            goto bad_cleanup_mmap;
        }
```

```
      //拷贝内容到进程空间中去
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memcpy(page2kva(page) + off, from, size);
            start += size, from += size;
        }
```

```

      //(3.6.2) 清空BSS段
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);// 用0填充
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }
```

```
    //(4) 创建栈
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags,
NULL)) != 0) {// 用户栈
        goto bad_cleanup_mmap;
    }
```

```
    //(5) 加载设置好的页表
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));
```

```
    //(6) 为用户环境设置 trapframe
struct trapframe *tf = current->tf;  // 获取当前进程的陷阱帧结构体指针
// Keep sstatus
uintptr_t sstatus = tf->status;  // 保存当前陷阱帧中的状态寄存器值
memset(tf, 0, sizeof(struct trapframe));  // 清空陷阱帧结构体

// Set user stack top
tf->gpr.sp = USTACKTOP;  // 设置用户栈顶，即陷阱帧中的用户栈指针

// Set entry point of user program
tf->epc = elf->e_entry;  // 设置用户程序的入口地址，即陷阱帧中的程序计数器

// Set status register appropriately
// tf->status should be appropriate for the user program (the value of
sstatus)
tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
// 在状态寄存器中设置 SPP 和 SPIE 的值，确保陷阱帧中的状态适合用户程序执行
// SSTATUS_SPP：用户态栈指针指向的是用户态栈（0），而不是内核态栈（1）
// SSTATUS_SPIE：允许中断和异常的外部中断
```

练习2: 父进程复制自己的内存空间给子进程（需要编码）

创建子进程的函数`do_fork`在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过`copy_range`函数（位于kern/mm/pmm.c中）实现的，请补充`copy_range`的实现，确保能够正确执行。

请在实验报告中简要说明你的设计实现过程。

```
            // 如果启用写时复制
            if(share)
            {
                cprintf("Sharing the page 0x%x\n", page2kva(page));
                // 物理页面共享，并设置两个PTE上的标志位为只读
                page_insert(from, page, start, perm & ~PTE_W);
                ret = page_insert(to, page, start, perm & ~PTE_W);
```

```
            }
            // 完整拷贝内存
            else
            {
                // alloc a page for process B
                // 目标页面地址
                struct Page *npage = alloc_page();
                assert(page!=NULL);
                assert(npage!=NULL);
                cprintf("alloc a new page 0x%x\n", page2kva(npage));
                void * kva_src = page2kva(page);
                void * kva_dst = page2kva(npage);
                memcpy(kva_dst, kva_src, PGSIZE);
                // 将目标页面地址设置到PTE中
                ret = page_insert(to, npage, start, perm);
            }
```

1. **页表项获取**：
   - 使用 `get_pte` 函数获取进程 A 中给定线性地址 `start` 处的页表项 `ptep`。
   - 如果 `ptep` 为 `NULL`，说明当前地址不在页表中，可能跨越了一个页表，需要对 `start` 进行调整。
2. **页表项创建**：
   - 使用 `get_pte` 函数获取进程 B 中给定线性地址 `start` 处的页表项 `nptep`。
   - 如果 `nptep` 为 `NULL`，说明进程 B 的页表中没有相应的项，需要创建一个新的页表项。
3. **页面复制**：
   - 获取进程 A 中页面的指针 `page`。
   - 根据写时复制标志

     ```
     share
     ```

     的值，执行不同的操作：
     - 如果启用写时复制，共享页面，并设置两个 PTE 上的标志位为只读。
     - 如果不启用写时复制，分配一个新的页面 `npage`，并将源页面的内容复制到目标页面。
4. **页面插入**：
   - 使用 `page_insert` 函数将新页面 `npage` 映射到进程 B 的地址空间的 `start` 处。

- 如何设计实现`Copy on Write`机制？给出概要设计，鼓励给出详细设计。

  资源分配： 当进程创建时，将其内存空间标记为“共享”状态。这可以通过将相应的页面表项设置为只读来实现。

  写操作触发： 如果一个进程试图写入一个已经标记为共享的内存页，会触发一个异常。

  拷贝操作： 在写操作触发时，内核会为写操作的进程创建一个独立的拷贝（副本）。

  更新页表： 更新页表，使得写操作的进程指向新创建的拷贝，而其他进程仍然指向原始共享的内存。

> Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

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（不需要编码）

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

- 请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？

  用户态调用的exec(), 归根结底是do_execve()

```
void
exit(int error_code) {
    sys_exit(error_code);
    cprintf("BUG: exit failed.\n");
    while (1);
}

int
fork(void) {
    return sys_fork();
}

int
wait(void) {
    return sys_wait(0, NULL);
}

```

  这里把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数，完成了系统调用的转发。接下来就是在`do_exit()，do_execve()`等函数中进行具体处理了。

```
static int
sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}

static int
sys_fork(uint64_t arg[]) {
    struct trapframe *tf = current->tf;
    uintptr_t stack = tf->gpr.sp;

```

```
    return do_fork(0, stack, tf);
}

static int
sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
    return do_wait(pid, store);
}

static int
sys_exec(uint64_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    return do_execve(name, len, binary, size);
}
```

```
int
sys_exit(int64_t error_code) {
    return syscall(SYS_exit, error_code);
}

int
sys_fork(void) {
    return syscall(SYS_fork);
}

int
sys_wait(int64_t pid, int *store) {
    return syscall(SYS_wait, pid, store);
}
```

在用户态进行系统调用的核心操作是，通过内联汇编进行`ecall`环境调用。这将产生一个trap, 进入S mode进行异常处理。

```
static inline int
syscall(int64_t num, ...) {
    va_list ap;
    va_start(ap, num);
    uint64_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i ++) {
        a[i] = va_arg(ap, uint64_t);
    }
    va_end(ap);

    asm volatile (
        "ld a0, %1\n"
```

```
        "ld a1, %2\n"
        "ld a2, %3\n"
        "ld a3, %4\n"
        "ld a4, %5\n"
        "ld a5, %6\n"
        "ecall\n"
        "sd a0, %0"
        : "=m" (ret)
        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
        :"memory");
    return ret;
}
```

- 请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

```
  +------------------+
  |     Created      |
  |                  |
  |     fork()       |
  +---------+--------+
            |
            v
  +---------+--------+
  |     Ready        |
  |                  |
  |    execve()      |
  +---------+--------+
            |
            v
  +---------+--------+
  |   Running        |
  |                  |
  | System Call /    |
  |  Timer Interrupt |
  +---------+--------+
            |
            v
  +---------+--------+
  |     Blocked      |
  |                  |
  | I/O Operation /  |
  |   Wait for Event |
  +---------+--------+
            |
            v
  +---------+--------+
  |    Exit          |
```

```
   |                  |
   |  exit() / Error  |
   +------------------+

```

1. **Created:**
   - 进程被创建，尚未开始执行。
   - 触发事件：`fork()`。
2. **Ready:**
   - 进程已准备好运行，等待 CPU 调度。
   - 触发事件：`execve()`。
3. **Running:**
   - 进程正在执行中，可能是执行用户程序的指令，也可能是在执行系统调用或被定时器中断唤醒。
   - 触发事件：系统调用、定时器中断。
4. **Blocked:**
   - 进程由于等待某些事件（如 I/O 操作完成）而被阻塞。
   - 触发事件：I/O 操作、等待事件。
5. **Exit:**
   - 进程执行完毕，或者由于发生错误而退出。
   - 触发事件：`exit()` 或者程序发生错误。

扩展练习 Challenge

1. 说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

用户程序是加载ucore的时候被一起加载进内存的，常用的操作系统是通过作业调度把用户程序从后备队列中加载进内存。原因是还没有实现文件系统，不存在外存。