Lab2 用户程序

石爽 马思远 薄照轩

练习1：加载应用程序并执行（需要编码）

设计实现过程

load_icode函数的核心目标是为用户程序构建合法的用户内存空间，并配置trapframe以确保进程能正确从用户态启动执行。第6步的关键是设置trapframe的三个核心字段，使其符合用户态执行要求：

1. **用户栈指针（tf->gpr.sp）**：用户栈空间已通过mm_map映射到USTACKTOP - USTACKSIZE至USTACKTOP区间，因此直接设置为USTACKTOP（用户栈顶地址），确保用户程序能正常使用栈空间。
2. **程序入口地址（tf->epc）**：ELF文件的e_entry字段存储了应用程序的起始执行地址，将其赋值给epc，使得CPU从该地址开始执行用户代码。
3. **状态寄存器（tf->status）**：清除SSTATUS_SPP位（设为0），表示返回用户态；设置SSTATUS_SPIE位，允许用户态响应中断；清除SSTATUS_SIE位，避免中断干扰当前状态切换，最终状态为`(sstatus & ~(SSTATUS_SPP | SSTATUS_SIE)) | SSTATUS_SPIE`。

代码实现

```
// setup trapframe for user environment
struct trapframe *tf = current->tf;
// Keep sstatus
uintptr_t sstatus = tf->status;
memset(tf, 0, sizeof(struct trapframe));
// 设置用户栈指针：用户栈顶部
tf->gpr.sp = USTACKTOP;
// 设置程序计数器：ELF入口地址
tf->epc = elf->e_entry;
// 设置状态寄存器：清除SPP（用户态），启用SPIE（用户态中断），禁用SIE
tf->status = (sstatus & ~(SSTATUS_SPP | SSTATUS_SIE)) | SSTATUS_SPIE;
```

从RUNNING态到执行应用程序第一条指令的经过

1. 调度器通过`schedule()`函数选择处于RUNNABLE态的用户进程，调用`proc_run()`切换当前进程为该用户进程。
2. `proc_run()`执行上下文切换：保存前一个进程的上下文，加载当前用户进程的`context`，其中`context.ra`指向`forkret`函数，`context.sp`指向进程的trapframe。
3. 执行`forkret()`函数，直接跳转到`__trapret`，触发中断返回流程。
4. `__trapret`执行`RESTORE_ALL`宏，恢复trapframe中的寄存器值，包括sp（用户栈顶）、epc（程序入口）和status（用户态权限）。
5. 执行`sret`指令，CPU特权级从内核态（S态）切换到用户态（U态），并跳转到epc指向的ELF入口地址。
6. 最终，CPU开始执行应用程序的第一条指令。

练习2：父进程复制自己的内存空间给子进程（需要编码）

设计实现过程

copy_range函数负责将父进程用户地址空间中[start, end)区间的内容拷贝到子进程，核心是按页复制，确保子进程拥有独立的内存空间：

1. **遍历地址范围**：以PGSIZE为步长，遍历父进程的目标地址区间，逐个处理每个页面。
2. **获取父进程页表项**：通过`get_pte()`函数找到父进程当前地址对应的页表项（ptep），确认页面是否有效（PTE_V）。
3. **分配子进程页面**：为子进程分配新的物理页面（npage），确保内存独立性。
4. **复制页面内容**：通过`page2kva()`将父进程页面和子进程页面转换为内核虚拟地址，使用`memcpy()`复制整页内容（PGSIZE字节）。
5. **建立子进程地址映射**：调用`page_insert()`函数，将子进程的物理页面与目标线性地址建立映射，权限与父进程保持一致（PTE_USER相关权限）。

代码实现

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share)
{
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    do {
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            struct Page *page = pte2page(*ptep);
            struct Page *npage = alloc_page();
            assert(page != NULL && npage != NULL);
            // 复制页面内容
            void *src_kvaddr = page2kva(page);
            void *dst_kvaddr = page2kva(npage);
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
            // 建立子进程地址映射
            int ret = page_insert(to, npage, start, perm);
            if (ret != 0) {
                return ret;
            }
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

Copy on Write（COW）机制设计

概要设计

COW的核心是"读共享、写复制"，避免fork时的冗余拷贝，提升效率：

1. fork时，父子进程共享父进程的物理页面，不复制实际内容。
2. 共享页面的页表项设置为只读权限（清除PTE_W）。
3. 当任一进程尝试写入共享页面时，触发页错误（Page Fault）。
4. 内核处理页错误时，为写入进程分配新的物理页面，复制原页面内容，更新页表项为可写，实现"私有拷贝"。

详细设计

1. **页表项标记与权限设置**：
    - fork时，在`copy_mm`中修改逻辑：不调用`copy_range`复制页面，而是直接共享父进程的mm_struct，增加页面引用计数。
    - 将父子进程共享页面的页表项权限设为`PTE_V | PTE_R | PTE_U`（只读、用户可访问），清除PTE_W位。
    - 新增页面引用计数字段（如`page->ref`），初始时父子进程共享页面的引用计数为2。
2. **页错误处理逻辑**：
    - 在`trap.c`的异常处理函数中，新增对"写只读页面"的页错误类型判断（通过stval寄存器获取错误地址，检查页表项权限）。
    - 若确认是COW触发的页错误，执行以下操作：
        1. 分配新的物理页面（`alloc_page()`）。
        2. 复制原页面内容到新页面（`memcpy(page2kva(new_page), page2kva(old_page), PGSIZE)`）。
        3. 减少原页面的引用计数，若计数为0则释放原页面。
        4. 更新当前进程的页表项，将错误地址映射到新页面，并恢复PTE_W权限（`PTE_V | PTE_R | PTE_W | PTE_U`）。
        5. 清除页错误标志，返回用户态继续执行写入操作。
3. **引用计数管理**：
    - 页面分配时，引用计数初始化为1。
    - fork共享页面时，引用计数加1。
    - 进程退出或页面解除映射时，引用计数减1，计数为0时释放页面。

练习3：分析fork/exec/wait/exit的实现及系统调用

函数执行流程分析

1. fork：创建子进程

- **用户态操作**：调用`fork()`函数（用户库封装），通过内联汇编执行`ecall`指令，触发系统调用，传递SYS_fork编号。

- **内核态操作**：
  1. 中断处理程序识别到USER_ECALL，调用`syscall()`函数，根据编号转发到`sys_fork`。
  2. `sys_fork`调用`do_fork()`，完成子进程创建：
     - 分配并初始化子进程的proc_struct（设置父进程、PID等）。
     - 共享或复制父进程的内存空间（`copy_mm`），通过`copy_range`复制用户内存（无COW时）。
     - 复制父进程的trapframe和上下文，子进程的a0寄存器设为0（标识子进程）。
     - 将子进程状态设为RUNNABLE，加入调度队列。
  3. 内核态执行完成后，将子进程PID写入父进程的a0寄存器，作为返回值。
- **结果返回**：通过`trapframe`中的a0寄存器传递返回值，父进程返回子进程PID，子进程返回0。

2. exec：加载新程序替换当前进程

- **用户态操作**：调用`exec()`函数（用户库封装），传递程序名称、二进制数据等参数，触发SYS_exec系统调用。
- **内核态操作**：
  1. `sys_exec`转发到`do_execve()`，回收当前进程的用户内存空间（`exit_mmap`、`put_pgdir`）。
  2. 调用`load_icode()`，解析ELF文件，为新程序创建内存空间，映射代码段、数据段、BSS段和用户栈。
  3. 重新设置trapframe，更新epc为新程序的ELF入口地址。
- **结果返回**：若执行成功，新程序直接覆盖当前进程，无返回值；若失败，返回错误码。

3. exit：进程退出

- **用户态操作**：调用`exit()`函数（用户库封装），传递退出码，触发SYS_exit系统调用。
- **内核态操作**：
  1. `sys_exit`转发到`do_exit()`，释放当前进程的用户内存空间（`exit_mmap`、`put_pgdir`）。
  2. 将进程状态设为PROC_ZOMBIE，保存退出码，唤醒父进程。
  3. 处理子进程继承：将当前进程的子进程托付给initproc。
  4. 调用`schedule()`切换到其他进程，当前进程不再执行。
- **结果返回**：无返回值，进程直接终止。

4. wait：父进程等待子进程退出

- **用户态操作**：调用`wait()`或`waitpid()`，触发SYS_wait系统调用，传递子进程PID（可选）。
- **内核态操作**：
  1. `sys_wait`转发到`do_wait()`，遍历父进程的子进程。
  2. 若存在ZOMBIE状态的子进程，回收其内核栈和proc_struct，返回退出码。
  3. 若无子进程或子进程未退出，将父进程状态设为PROC_SLEEPING（WT_CHILD），调用调度器切换进程。
  4. 当子进程退出唤醒父进程后，重复上述步骤。
- **结果返回**：成功返回子进程PID，退出码通过参数传递；失败返回错误码。

内核态与用户态的交错执行及结果返回

- **交错执行**：用户程序通过`ecall`指令触发系统调用，从用户态切换到内核态；内核处理完成后，通过`sret`指令返回用户态，继续执行用户程序。

- **结果返回**：内核态通过修改`trapframe`中的a0寄存器传递返回值，用户程序从`ecall`指令的下一条指令继续执行，读取a0寄存器获取结果。

用户态进程执行状态生命周期图

```
PROC_UNINIT
    |
    +-- alloc_proc() 初始化
    v
PROC_RUNNABLE
    |
    +-- 调度器选中 (proc_run())
    v
RUNNING
    |
    +-- 时间片耗尽/主动yield (do_yield())
    |    +--> PROC_RUNNABLE
    |
    +-- 调用wait()/sleep() (do_wait()/do_sleep())
    |    +--> PROC_SLEEPING
    |             |
    |             +-- 被唤醒 (wakeup_proc())
    |             +--> PROC_RUNNABLE
    |
    +-- 调用exit() (do_exit())
    |    +--> PROC_ZOMBIE
    |             |
    |             +-- 父进程wait() (do_wait())
    |             +--> 资源回收 (kfree/proc_struct)
    |
    +-- 被kill() (do_kill())
         +--> PROC_ZOMBIE
              |
              +-- 父进程wait()
              +--> 资源回收
```

测试结果

执行`make grade`，所有应用程序检测均输出“ok”，表明实验代码正确实现了用户进程的加载、创建、退出和等待功能，符合实验要求。

```
root@Ss:~/labcode/lab5# make grade
  -check result:                             OK
  -check output:                             OK
faultread:                 (1.0s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:           (1.0s)
  -check result:                             OK
  -check output:                             OK
hello:                     (1.0s)
  -check result:                             OK
  -check output:                             OK
testbss:                   (1.0s)
  -check result:                             OK
  -check output:                             OK
pgdir:                     (1.0s)
  -check result:                             OK
  -check output:                             OK
yield:                     (1.0s)
  -check result:                             OK
  -check output:                             OK
badarg:                    (1.0s)
  -check result:                             OK
  -check output:                             OK
exit:                      (1.0s)
  -check result:                             OK
  -check output:                             OK
spin:                      (4.2s)
  -check result:                             OK
  -check output:                             OK
forktest:                  (1.0s)
  -check result:                             OK
  -check output:                             OK
Total Score: 130/130
```