Week6 Report in Class (Fri56)

> 11911839 聂雨荷

Q1

> 代码中通过何种方式从S mode 进入U mode?

1. Create one process execute function `user_main()` in `init_main()` function we have discussed in the previous lab
   - We create `proc_struct` for the new process, and add it to the process linked list

```
static int init_main(void *arg) {
    size_t nr_free_pages_store = nr_free_pages();
    //size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    ...
}
```

2. As the administer, `init_main()`, will call `do_wait` to wait for the ending of the child process, and use `schedule()` to schedules `user_main()` during the wait

```
static int init_main(void *arg) {
    ...

    while (do_wait(0, NULL) == 0) {
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->optr == NULL);
    assert(nr_process == 2);
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));

    cprintf("init check memory pass.\n");
    return 0;
}
```

3. As we have discussed in Lab5, `schedule()` function will create find the `user_main()` process and execute it. `user_main()` calls the `KERNEL_EXECVE()` function

```c
// user_main - kernel thread used to exec a user program
static int user_main(void *arg) {
    KERNEL_EXECVE(hello);
    panic("user_main execve failed.\n");
}
```

```c
#define __KERNEL_EXECVE(name, binary, size) ({                          \
            cprintf("kernel_execve: pid = %d, name = \"%s\".\n",        \
                    current->pid, name);                                \
            kernel_execve(name, binary, (size_t)(size));                \
        })

#define KERNEL_EXECVE(x) ({                                             \
            extern unsigned char _binary_obj___user_##x##_out_start[],  \
                _binary_obj___user_##x##_out_size[];                    \
            __KERNEL_EXECVE(#x, _binary_obj___user_##x##_out_start,     \
                            _binary_obj___user_##x##_out_size);         \
        })
```

- This two function later will allows `user_main()` to execute the code `hello.c`

```c
kern_execve("hello",
_binary_obj___user_hello_out_start,_binary_obj___hello_exit_out_size)
```

4. `kern_execve()` triggers ebreak exception

```c
// kernel_execve - do SYS_exec syscall to exec a user program called by user_main
kernel_thread
static int kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
    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" // trigger exception
        "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;
}
```

5. This ebreak exception will be caught by `trap()` function. `trap()` function recognizes the cause is `CAUSE_BREAKPOINT` and enters `syscall()` to handle this problem

```c
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        ...
        case CAUSE_BREAKPOINT:
            cprintf("Breakpoint\n");
```

```
            if(tf->gpr.a7 == 10){
                //tf->epc += 4;
                syscall();
            }
            break;
        ...
    }
}
```

6. `syscall()` calls the corresponding function defined by an array of syscalls, in this case, register `a0` has been set to be `4`, which is `SYS_exec`, continuing execute `SYSexec()`

```
void
syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0;
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            return ;
        }
    }
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, current->pid, current->name);
}


static int (*syscalls[])(uint64_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_getpid]            sys_getpid,
    [SYS_putc]              sys_putc,
};
```

7. `sys_exec()` calls `do_excve()`

```
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);
}
```

8. `do_excve()` allocates required memory for the new process

```c
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of current process
//           - call load_icode to setup new memory space accroding binary prog.
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) { // prepare for the code
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

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

9. `load_icode()` function sets the required resource. Specifically, it will set the `tf->status` to `0`, which will wait for the setting of the operating system

```c
/* load_icode - load the content of binary program(ELF format) as the new content of current process
 * @binary:  the memory addr of the content of binary program
 * @size:  the size of the content of binary program
 */
static int
load_icode(unsigned char *binary, size_t size) {
    ...
    tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
    ...
    goto out;
}
```

10. after the `trap` finish loading the process, it will call the following code to return. This `sret` is a RISV command and sets `CSRs[sstatus].SPP = 0`, a User mode

```
# trapentry.S
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret

    .globl forkrets
```

Q2

> 代码中用户进程调用系统调用的过程是怎样的?

1. Initially, `hello.c` uses `cprintf()` to print the string

```
#include <stdio.h>
#include <ulib.h>

int
main(void) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n", getpid());
    cprintf("hello pass.\n");
    return 0;
}
```

2. `cprintf()` encapsulates several functions. It finally calls the `sys_putc()` rather than call `sbi_console_putchar()`. This is because only S mode can use `ecall` to call the OpenSBI interface. We have to use system call.

```
/* *
 * cprintf - formats a string and writes it to stdout
 *
 * The return value is the number of characters which would be
 * written to stdout.
 * */
int cprintf(const char *fmt, ...) {
    va_list ap;

    va_start(ap, fmt);
    int cnt = vcprintf(fmt, ap); // call vcprintf
    va_end(ap);

    return cnt;
}

/* *
 * vcprintf - format a string and writes it to stdout
 *
 * The return value is the number of characters which would be
 * written to stdout.
 *
 * Call this function if you are already dealing with a va_list.
```

```
 * Or you probably want cprintf() instead.
 * */
int vcprintf(const char *fmt, va_list ap) {
    int cnt = 0;
    vprintfmt((void*)cputch, &cnt, fmt, ap); // call cputch
    return cnt;
}

/* *
 * cputch - writes a single character @c to stdout, and it will
 * increace the value of counter pointed by @cnt.
 * */
static void cputch(int c, int *cnt) {
    sys_putc(c);
    (*cnt) ++;
}
```

3. `sys_putc()` calls `syscall()` transform from U mode to S mode.

- `sys_call()` function makes `ecall` environment calls through inline assembly. This will generate a trap and enter S mode for exception handling

```
int sys_putc(int64_t c) {
    return syscall(SYS_putc, c);
}

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;
}
```

4. Trap will be captured by `trap.c` and forward this system call to `syscall()` function defined in the kernel

```c
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        ...
        case CAUSE_USER_ECALL:
            //cprintf("Environment call from U-mode\n");
            tf->epc += 4;
            syscall();
            break;
        ...
    }
}
```

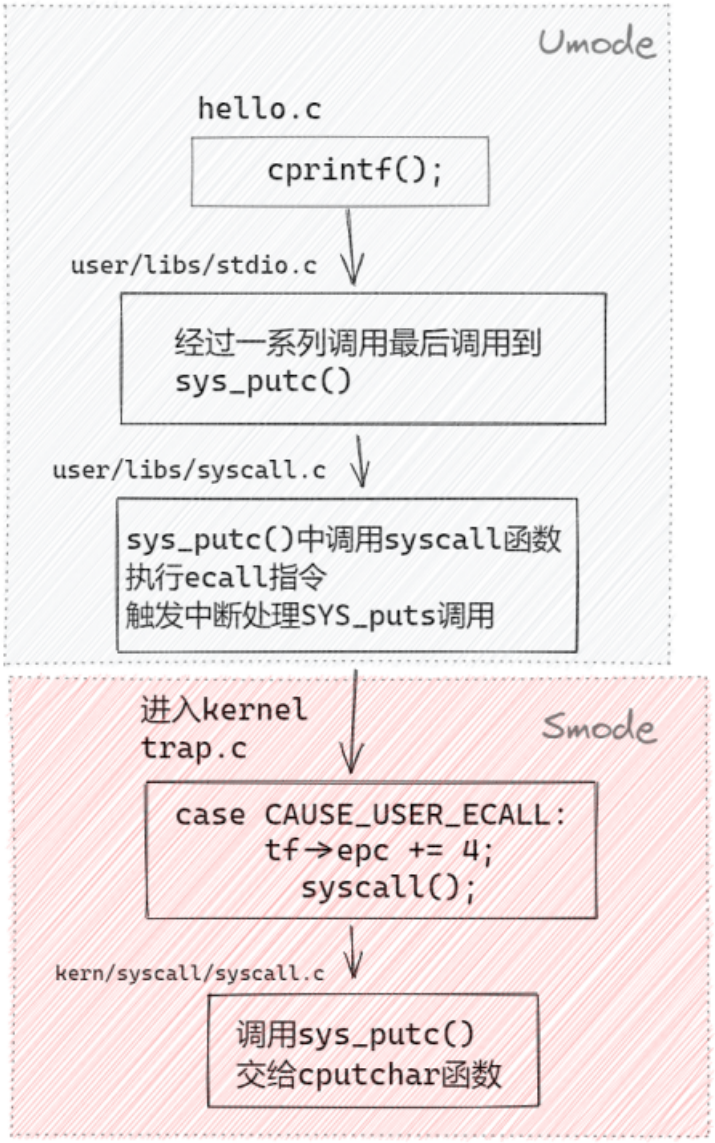
5. `syscall()` takes the arguments in the register and forwards them to the function corresponding to the system call number for processing

```c
void syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0;
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            return ;
        }
    }
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, current->pid, current->name);
}
```

6. `syscall()` will trigger `sys_putc()`, as the user call requests system call to print the string

```c
static int sys_putc(uint64_t arg[]) {
    int c = (int)arg[0];
    cputchar(c);
    return 0;
}
```

7. We have discussed `cputchar()` function in Assignment2. It will trigger OpenSBI function. At this point, we have completed the system call.



Q3

> 代码中 `hello.c` 的 main() 执行结束后发生了什么，模式是否切换?

1. When the `hello.c` in the user mode finishes its execution, the process finishes its work. `hello.c` is actually called by `umain.c`, as the following code shows, `umain` will call the `exit(ret)`

```
void
umain(void) {
    int ret = main();
    exit(ret);
}
```

2. `exit(ret)` calls `sys_exit`, which is a system call

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

3. the rest of the procedures are like basic system call procedure we have discussed in the previous lab

- command `ecall` causes a trap, which will be caught and deliver to the kernel. At this time, **User mode is transformed into Kernel mode**

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

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" // ecall, execute system call in the kernel mode
        "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;
}


// handle by the kernel system call function
static int
sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}

```

4. `do_exit()` recycles the user process

- Set the state of current process to be `PROC_ZOMBIE`. Waiting for the parent process to recycle the process. Set the `current->exit_code = error_code`; means that this process can not be scheduled
- If the parent of the current process is waiting for a child, wake up the parent to reclaim resources

- Then start the interrupt, execute the `schedule()` function, and select a new process to execute

```
// do_exit - called by sys_exit
//   1. call exit_mmap & put_pgdir & mm_destroy to free the almost all memory space of process
//   2. set process' state as PROC_ZOMBIE, then call wakeup_proc(parent) to ask parent reclaim itself.
//   3. call scheduler to switch to other process
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }
    struct mm_struct *mm = current->mm;
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;
    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);
    schedule();
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

> 首先执行 `lcr3(boot_cr3);` 切换到内核的页表上，这样用户进程就只能在内核的虚拟地址空间上执行，因为内核权限高。如果当前进程的被调用数减一后等于`0`，那么就没有其他进程在使用了，就可以进行回收，先回收内存资源，调用 `exit_mmap` 函数释放 `mm` 中的 `vma` 描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后把页表项和页目录表清空。然后调用 `put_pgdir` 函数释放页目录表所占用的内存。最后调用 `mm_destroy` 释放 `vma` 与 `mm` 的内存。把`mm` 置为`NULL`，表示与当前进程相关的用户虚拟内存空间和对应的内存管 理成员变量所占的内核虚拟内存空间已经回收完毕

Q4

> 进程如何变成僵尸进程?

After child process finishes its execution. The state of child process will be set `PROC_ZOMBIE`. Indicates that the child process is in the exit state and requires current process (parent process) complete the final reclamation of the child process. If the current process is still in execution and has not been executed to `wait` to reclaim the child process, then this child process will become the zombie process.