Lab6 Exec and Wait

0. 前言

实验概述

支持exec与wait系统调用，简单了解设备输入，最后支持Shell

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab6-exec-wait
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab6-exec-wait
```

本次代码新增系统调用：

1. SYS_READ：用于读取用户输入
2. SYS_SCHED_YIELD：用于用户进程放弃当前时间片
3. SYS_EXECVE：Exec系统调用支持，用于切换至另一个程序
4. SYS_WAIT4：Wait系统调用支持，用于等待子进程退出

1. Exec

在上节内容中，我们讲解了多进程系统，并实现了一版简单的Fork系统调用，但该系统调用只会不断地运行单个程序，带来了比较大的限制，为了解决该问题，我们需要进一步支持Exec系统调用，以运行其他程序，

Exec系统调用会根据**用户传入的路径**，读取该路径下的文件内容，并加载**覆盖当前进程的用户空间**，以达到运行其他程序的目的，将初始程序更改为`exec`，我们能看到以下输出：

```
My PID: 1
This is before exec
Hello World!, my pid: 1
```

在`exec.c`代码的最后，我们尝试输出`This is after exec`，但在实际的输出中没有看到这句话，原因在于调用exec之后该进程完全会放弃当前程序运行，转而重新加载并运行另外一个程序：`execl("hello_world", "hello_world", "temp",*NULL);`

上面的代码调用了exec，并传入了`hello_world`字符串，指示内核查找`hello_world`程序是否存在，如存在则切换到该程序运行。内核接收到该字符串后，会调用`progs::lookup_progs`来查找要运行的二进制程序（简易key-value存储）。如存在，内核便会解析该程序，并覆盖当前地址空间**。需要注意的是进程的父子关系，pid值等并不会发生改变。**

内核具体实现exec的代码如下：

```rust
// src/syscall/exec.rs
pub fn sys_execve(
    path: Vaddr, /* &[u8] */
    argv: Vaddr, /* &[&str] */
    envp: Vaddr, /* &[&str] */
    current_process: &Arc<Process>,
    user_context: &mut UserContext,
) -> Result<SyscallReturn> {
    ...

    let binary = crate::progs::lookup_progs(exec_name)?;
    *user_context = current_process.exec(binary);

    Ok(SyscallReturn(0 as _))
}


// src/process/mod.rs
pub fn exec(&self, binary: &[u8]) -> UserContext {
    self.memory_space.clear();
    elf::load_user_space(binary, &self.memory_space)
}

// src/process/elf.rs
pub fn load_user_space(program: &[u8], memory_space: &MemorySpace) -> UserContext
{
    let mut user_context = UserContext::default();
    parse_elf(program, &memory_space, &mut user_context);
    user_context
}
```

可以看到其代码十分简单，只进行了：

1. 清空当前进程的用户空间
2. 根据字节数组，解析ELF文件
3. 设置用户空间，并返回准备好的用户上下文（程序寄存器）
4. 重设用户上下文（程序寄存器）

2. Wait

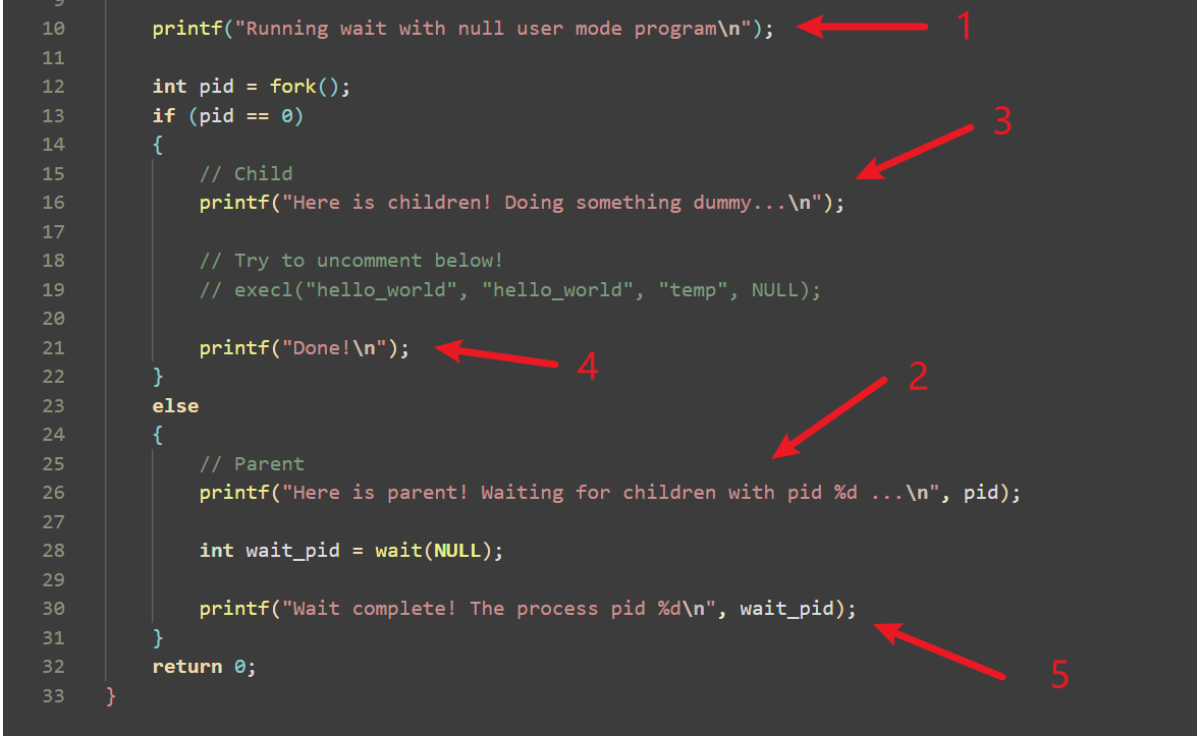
Exec与Fork一起合作，能让一个用户进程不断地创建子进程，并运行存在于操作系统的其他程序，两者协同既可实现从1到多的系统多功能进程创建，但某些情况下我们要进行的下一步操作依赖于子进程的完成，比如经典的Shell，此时就需要多引入一个系统调用，该系统调用需要实现：**等待子进程退出**的功能。该系统调用便是`wait`，在代码中为`SYS_WAIT4`

首先观察下wait系统调用会带来什么输出，将初始程序更改为`wait`，我们能看到以下输出：

```
Running wait with null user mode program
Here is parent! Waiting for children with pid 2 ...
Here is children! Doing something dummy...
Done!
Wait complete! The process pid 2
```

对应到代码中，我们能看到以下的输出顺序：

```c
int main(int argc, char *argv[])
{
    // Disable buffer in STDOUT
    setvbuf(stdout, NULL, _IONBF, 0);

    printf("Running wait with null user mode program\n");

    int pid = fork();
    if (pid == 0)
    {
        // Child
        printf("Here is children! Doing something dummy...\n");

        // Try to uncomment below!
        // execl("hello_world", "hello_world", "temp", NULL);

        printf("Done!\n");
    }
    else
    {
        // Parent
        printf("Here is parent! Waiting for children with pid %d ...\n", pid);

        int wait_pid = wait(NULL);

        printf("Wait complete! The process pid %d\n", wait_pid);
    }
    return 0;
}
```



可以看到parent在进行fork操作，因`wait`的存在，不会立即打印出最后的`Wait complete`，而是会切换到子进程运行，并在子进程退出后，才会输出`Wait complete ...`，这样就达到了等待目标进程退出的目标，该目标是我们接触的第一个 **等待-唤醒** 功能，我们先来看看实现`wait`系统调用的关键代码：

```rust
pub fn wait(&self, wait_pid: i32) -> Result<(Pid, u32)> {
    let wait_pid = if wait_pid == -1 {
        None
    } else {
        Some(wait_pid.abs() as Pid)
    };
    let res = self.try_wait(wait_pid);

    match res {
        Ok((pid, exit_code)) => return Ok((pid as Pid, exit_code)),
        Err(err) if err.code == Errno::EAGAIN => {}
        Err(err) => return Err(err),
    }

    // No child exit, waiting...
    let wait_queue = &self.wait_children_queue;
    Ok(wait_queue.wait_until(|| self.try_wait(wait_pid).ok()))
}
```

解析目标：-1 表示等“任意子进程”，否则等指定 pid

先做一次“非阻塞检查”（快速路径）

找到已退出（Zombie）的子进程，直接返回（不用睡眠）

EAGAIN 表示“现在还没有可回收的子进程”，不是致命错误——继续走阻塞等待

其它错误（比如没有子进程 ECHILD、目标 pid 不是子进程等）直接返回

真正的阻塞等待路径：挂到等待队列，直到条件满足

该函数会首先进行尝试操作，如果没有子进程退出，则会进行最后的等待，其会调用`wait_queue.wait_until()`，该函数会让当前进程阻塞，直到有 **其他进程唤醒，并条件满足** 时才会返回。为正确实现wait，我们需要确定所说的两件事情：（1）唤醒位置；（2）停止等待条件。

1. 唤醒位置：我们的目标是实现**等待指定子进程退出**，因此最好的唤醒位置便是`exit`，在`Process::exit`可以看到：

```
// Wakeup the parent process if it is waiting.
if let Some(parent) = self.parent_process() {
    parent.wait_children_queue.wake_all();
}
```

2. 停止等待条件：判断是否有子进程退出（为Zombie状态），且其Pid是否为目标Pid，相关逻辑实现在了`Process::try_wait`：

```
for (child_pid, child) in children.iter() {
    debug!(
        "try_wait: check child pid = {}, is zombie = {:?}",
        child_pid,
        child.status.is_zombie()
    );
    if child.status.is_zombie() {
        wait_pid = Some(*child_pid);
        break;
    }
}
```

这段代码来自 Process::try_wait()，用于检查是否有子进程已经退出（Zombie 状态）。

3. 上手练习

Init Process介绍

在之前的开发中，我们进行了一个比较烦人的重复操作，即每次要运行特定程序，都需要更改lib.rs中的初始程序，这对于开发和实际运行都不是很友好，因此我们最佳的方案就是将初始程序设置为一个固定的程序，并在该程序中利用Fork-exec-wait来运行指定的程序，这样就不用每次都需要修改初始程序了。该初始程序在我们的系统中，其功能有三个：

1. Fork子进程，并将子进程切换为Shell
2. 等待所有自己的子进程退出（以进行Reparent后的资源回收）
3. 当Shell退出时，主动调用`reboot(RB_POWER_OFF)`来退出系统

本次上手练习需要实现以上三个功能，即：（1）利用Fork-exec-wait来补充`shell.c`，使得可以运行用户输入的指定程序；（2）完善`reparent_children_to_init`函数，实现reparent功能；（3）**Optional**：实现`sys_reboot`系统调用，只实现`RB_POWER_OFF`即可，用exit_qemu来进行退出操作。

Simple Shell

简易Shell原理：在`wait.c`中包含了一句被注释掉的exec语句，取消注释可以得到一个fork-exec-wait一起使用的样例，如果将if改为while循环，并将exec执行的程序名改为输入指定，那么我们可以模拟一个非常简易的shell。

练习内容：本节课的代码中已经实现了`sys_read`系统调用，可以简单的从控制台读取字符内容，且提供了一个shell的基础框架：`shell.c`。大家需要通过fork-exec-wait这三个系统调用，实现一个其中的`execute`函数，完善shell的功能，示例截图：

```
Domain0 SysReset          : yes
Domain0 SysSuspend        : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : sstc,zicntr,zihpm,zicboz,zicbom,sdtrig,svadu
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 2 bits
Boot HART PMP Address Bits: 54
Boot HART MHPM Info       : 16 (0x0007fff8)
Boot HART Debug Triggers  : 2 triggers
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
Enter riscv_boot
[INIT] Starting Shell...
Running Shell...
~ # hello_world
 Running command: hello_world
Hello World!, my pid: 3
~ # exit
 Running command: exit
Exit shell
[INIT] Catch child process, pid: 2
[INIT] Shell process exited, exiting system...
```

Reparent

完善`Process::reparent_children_to_init`函数，运行`reparent`以进行测试（Hint：对于`Mutex<T>`，使用`.lock()`以拿到其中的可变引用），示例截图：

```
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : sstc,zicntr,zihpm,zicboz,zicbom,sdtrig,svadu
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 2 bits
Boot HART PMP Address Bits: 54
Boot HART MHPM Info       : 16 (0x0007fff8)
Boot HART Debug Triggers  : 2 triggers
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
Enter riscv_boot
[INIT] Starting Shell...
Running Shell...
~ # reparent                                          1
 Running command: reparent
[Reparent] Parent process exit, pid: 3
[Reparent] Child process, pid: 4, ppid: 3
[Reparent] Child process yielding, pid: 4
~ # hello_world                                       2
 Running command: hello_world
[Reparent] Child process reparenting to init, ppid: 1
Hello World!, my pid: 5
[INIT] Catch child process, pid: 4
~ # exit                                              3
 Running command: exit
Exit shell
[INIT] Catch child process, pid: 2
QEMU: Terminated
```

System Power OFF

实现`sys_reboot`系统调用中的**关闭系统分支**，系统调用号为142，相关手册：https://man7.org/linux/man-pages/man2/reboot.2.html，最终效果为shell退出后系统终止