0. 前言

实验概述

进程第二部分。运行实验代码，了解fork，exec与wait系统调用并更改进程运行顺序观察结果

实验内容

1. 拉取实验所需环境与代码
2. 了解实验代码的新架构
3. 运行与解析fork代码，修改内核的进程运行顺序，观察结果
4. 运行与解析exec代码
5. 运行与解析wait代码

1. 实验环境及代码

拉取实验所需环境以及代码：

```
podman pull glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0
mkdir os-lab
podman run -it -v ./os-lab:/root/os-lab glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0

git clone -b lab6 https://github.com/sdww0/sustech-os-lab.git
cd sustech-os-lab
```

运行实验代码:

```
cargo osdk run --scheme riscv --target-arch=riscv64
```

章节与对应示例程序：

1. Fork <-> `fork.c`
2. Exec <-> `exec.c`
3. Wait <-> `wait_with_null.c` & `wait_with_pid.c`

2. Fork-Exec-Wait

2.1 Fork

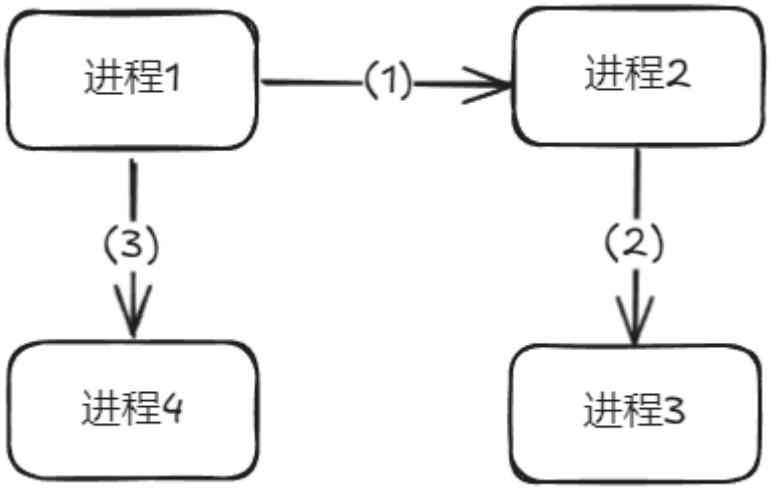
Fork可以为当前进程创建一个"副本"子进程，其几乎完全复制父进程的信息，但不同的是对于父进程fork会返回子进程的pid，而子进程只会返回0。在使用fork时可以根据这个信息来进行父子进程的区分。运行本节的示例代码，可以看到如下输出：

```
A1111, my pid: 1.
B2222, my pid: 2. val1: 0
B2222, my pid: 1. val1: 2
C3333, my pid: 3. val2: 0
C3333, my pid: 2. val2: 3
C3333, my pid: 4. val2: 0
C3333, my pid: 1. val2: 4
```

可以看到以`A1111`，`B2222`，`C3333`开头的输出分别有1，2，4次，这是因为示例代码里面总共有两次fork，其中第一次fork之后进程数由1变2，此时注意到子进程会完全复制父进程的信息，因此子进程也会运行第二次的fork，使得最终进程数会变成4个，表现为`C3333`输出了4次。

除了上面的信息，大家也可以注意到后面跟着的`val1`和`val2`。这两个信息也是比较重要的，其告诉了我们fork的返回值。对于val1输出了两次，分别为0和2，根据之前所说的内容，子进程的fork函数会返回0，父进程会返回子进程pid（>0），我们可以推断出**此时是子进程先运行，随后再运行了父进程**，该运行顺序是OSTD中的FIFO调度器决定的。

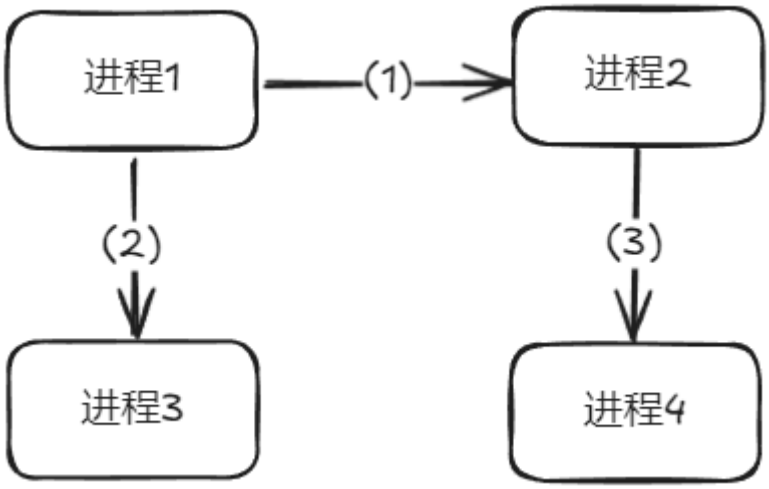
根据输出我们可以推导出这个程序的fork发生顺序以及父子进程关系图：



不同的调度策略，时钟中断发生时机会影响到该运行顺序，在`src/thread/mod.rs:61`行有一个注释，如果取消注释，会导致进程在创建时就会立马放弃当前时间片，使得即使是同一个用户态程序，上面的输出完全不一样。取消注释我们可以得到以下输出：

```
A1111, my pid: 1.
B2222, my pid: 1. val1: 2
B2222, my pid: 2. val1: 0
C3333, my pid: 1. val2: 3
C3333, my pid: 2. val2: 4
C3333, my pid: 3. val2: 0
C3333, my pid: 4. val2: 0
```

根据输出我们可以推导完全不同的fork发生顺序以及父子进程关系图：



大家可以尝试在Linux中多次运行`qemu-riscv64 ./target/user_prog/fork`来观察输出顺序，会得到和该系统不一样的结果。对于这个进程运行的具体解释在这里不会进行展开，感兴趣的同学可以自己推导一下原来的运行顺序是什么，取消注释后的运行顺序是什么。

2.2 Exec

在操作系统中存在另外一个系统调用exec，该系统调用可以与fork进行协助，使得某个进程去创建执行另一个二进制程序的子进程。运行本节代码可以看到输出

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

在代码中我们虽然在最后编写了一个`print`来输出一段话，但在输出中并没有这句，原因在于调用exec之后该进程完全会放弃当前二进制程序的运行，而去从零开始运行另外一个二进制程序。在`exec.c`中我们通过exec系统调用，直接运行了`hello_world`程序，**这个程序存放在我们的一个简易key-value存储中**。内核找到后便会解析该程序，并完全取代当前进程。需要注意的是进程的父子关系，pid值等并不会发生改变。

2.3 Wait

前面Fork实验的过程中，可以发现父子进程之间的执行顺序取决于操作系统的实现，并且在实际过程中会因为时钟中断使得同一个系统，同一个程序，运行结果却不一致。如果我们想要它们以固定的顺序执行，可以使用`wait()`。 一个实现了信号的系统中，当`wait()`执行时，当前进程会被挂起(suspend)，等待子进程结束。当子进程结束后会给父进程发送 SIGCHLD 信号，`wait()`收到信号后会结束阻塞并回收子进程的相关资源。

在本次的实验课代码中，并没有实现信号相关的机制，但是在PCB中我们有一个`wait_children_queue`，这个域是父进程用来等待子进程退出的一个等待队列，使用方式如下：当父进程调用`wait_children_queue.wait_until`时，如果不满足条件，便会进入到等待状态。当子进程退出时，会访问父进程的这个域，并进行唤醒操作。运行本节代码，可以观察到fork和wait是如何进行配合的，与Fork同样取消`src/thread/mod.rs:61`的注释，可以观察到不一样的结果。以取消注释后运行`wait_with_null`为例，其输出为

```
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
```

可以看到parent在进行fork操作后，会进行wait操作，等待子进程输出`Done!`并退出后，才会被唤醒并输出最后的语句。

Fork-Exec-Wait联动

在`wait_with_null.c`和`wait_with_pid.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。

3. 内部实现

3.1 Fork & Exec

Fork和Exec其实流程比较相似，因此它们放在了同一节来进行讲解。从代码实现上，它们均需要构建新的内存空间，写入对应的代码和数据，并设置对应的CPU寄存器。

3.1.1 Fork

在RISC-V中，Fork的底层系统调用实际上是使用的`sys_clone`，其代码是在`src/syscall/clone.rs`中，里面会根据用户态程序传入的`CloneFlags`和一系列其它的参数构建`CloneArgs`并将该参数传入`clone_current_process`。对于`CloneArgs`我们不需要进行关注，最需要了解的是`clone_current_process`中干了什么，以下是去除不需要了解内容后，该函数的代码：

```rust
pub fn clone_current_process(
    parent_context: &UserContext,
    clone_args: CloneArgs,
) -> Result<Arc<Process>> {
    let current_process = current_process().unwrap();

    // Clone VM
    let new_memory_space =
current_process.memory_space.duplicate_with_new_frames();

    // Clone User Context
    let mut context = *parent_context;
    context.set_a0(0);

    let process = Process::raw_new_user_process(
        context,
        new_memory_space,
        &current_process.heap,
        current_process.name(),
    );

    process.set_parent_process(&current_process);

    Ok(process)
```

```
}

```

可以看到，实际上纯fork的流程十分简单：（1）复制当前进程的内存空间；（2）复制当前进程的CPU状态，并设置返回值为0（注意，fork系统调用如果返回值为0则为子进程，不为0则为父进程）；（3）根据复制后的内容，创建子进程并设置其父进程为当前进程。

3.1.2 Exec

Exec的系统调用入口为`src/syscall/exec.rs`中的`sys_execve`，里面会读取用户态传入的可执行文件路径，并通过文件系统获取该文件。之后的`do_execve`会获取当前进程的内存空间，并传入`load_elf_to_vm_and_context`中，让我们来看该函数的代码：

```rust
pub fn load_elf_to_vm_and_context(
    program: &[u8],
    memory_space: &MemorySpace,
    context: &mut UserContext,
) {
    memory_space.clear();

    // Reset context
    let default_context = UserContext::default();
    *context.general_regs_mut() = *default_context.general_regs();
    context.set_tls_pointer(default_context.tls_pointer());
    *context.fp_regs_mut() = *default_context.fp_regs();

    parse_elf(program, memory_space, context);
}
```

首先，Exec需要清空当前进程的内存映射并重设当前进程的CPU寄存器状态，为之后重新加载新程序做准备。在`parse_elf`中，会解析指定的程序字节数组并构建内存空间和CPU寄存器，存放到传入的参数中。经过上述流程，该进程已经准备好运行新的程序了，只需直接返回到用户态即可。

3.2 Wait

Wait的系统调用入口为`src/syscall/wait4.rs`中的`sys_wait4`，其中我们关心的只有`wait_pid`一个域，其它的可以忽略。`wait_pid`里面会指定等待的子进程pid，如果为-1则代表任意的子进程退出即可。`sys_wait4`最后会调用`wait_child`来进行等待操作，源码如下：

```rust
pub fn wait_child(wait_pid: i32, process: Arc<Process>) -> Result<(Pid,
u32)> {
    if wait_pid == -1 {
        // Try first wait
        let res = process.wait_remove_one_nonblock();
        match res {
            Ok((pid, exit_code)) => return Ok((pid as Pid, exit_code)),
            Err(err) if err.error() == Errno::EAGAIN => {}
            Err(err) => return Err(err),
        }

        let wait_queue = process.wait_children_queue();
        Ok(wait_queue.wait_until(||
process.wait_remove_one_nonblock().ok()))
```

```
    } else {
        if wait_pid < -1 {
            warn!("We use pgid as pid since we don't support pgid");
        }

        let pid = wait_pid.abs();

        // Try first wait
        let res = process.wait_with_pid_nonblock(pid as u32);
        match res {
            Ok(exit_code) => return Ok((pid as Pid, exit_code)),
            Err(err) if err.error() == Errno::EAGAIN => {}
            Err(err) => return Err(err),
        }

        let wait_queue = process.wait_children_queue();
        let exit_code = wait_queue.wait_until(||
process.wait_with_pid_nonblock(pid as Pid).ok());

        Ok((pid as Pid, exit_code))
    }
}
```

两种情况会分别调用不同的函数，但首先均会进行第一轮等待，如果错误码为`EAGAIN`则代表子进程还没有退出。`wait_remove_one_nonblock`与`wait_with_pid_nonblock`差别是，一个会检查所有子进程是否退出，一个只会检查特定子进程是否退出。当两个分支中的检查失败后，会使用到`WaitQueue`等待队列来进行等待，由此将父进程进入到上节课提到的Wait状态。

当子进程退出时，会使用到`Process:exit`函数，其中会获取父进程的`wait_children_queue`并进行唤醒，当唤醒时父进程会再次检查是否符合等待条件，如果不满足（等待目标子进程时，其它子进程唤醒了父进程）还会继续睡眠，由此达到"等待-唤醒"的效果。

3.3 Reparent

本节代码还实现了Reparent机制，在`Process::reparent_children_to_init`中会将当前进程的所有子进程转移到初始进程上，并重新设置它们的父进程，代码如下：

```
pub fn reparent_children_to_init(&self) {
    const INIT_PROCESS_ID: Pid = 1;
    if self.pid == INIT_PROCESS_ID || self.children.lock().is_empty() {
        return;
    }

    // Do re-parenting
    let process_table = PROCESS_TABLE.lock();
    let init_process = process_table.get(&INIT_PROCESS_ID).unwrap();
    let mut init_process_children = init_process.children.lock();
    let mut self_children = self.children.lock();

    while let Some((pid, child)) = self_children.pop_first() {
        *child.parent_process.lock() = Arc::downgrade(init_process);
        init_process_children.insert(pid, child);
    }
}
```

4. 上手练习

Simple Shell!

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，大家可以通过fork-exec-wait这三个系统调用，实现一个简单的shell，运行其它C语言程序如`hello_world`...示例截图：

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
Enter riscv_boot
User programs: ["console_read", "exec", "fork", "hello_world", "shell", "wait_with_null", "wait_with_pid"]
Running simple shell!
~# wait_with_null
Running wait with null user mode program
Here is children! Doing something dummy...
Done!
Here is parent! Waiting for children with pid 3 ...
Wait complete! The process pid 3
~# fork
A1111, my pid: 4.
B2222, my pid: 5. val1: 0
B2222, my pid: 4. val1: 5
C3333, my pid: 6. val2: 0
C3333, my pid: 5. val2: 6
C3333, my pid: 7. val2: 0
C3333, my pid: 4. val2: 7
~# fork
A1111, my pid: 8.
B2222, my pid: 9. val1: 0
B2222, my pid: 8. val1: 9
C3333, my pid: 10. val2: 0
C3333, my pid: 9. val2: 10
C3333, my pid: 11. val2: 0
C3333, my pid: 8. val2: 11
~# exec
My PID: 12
This is before exec
Hello World!, my pid: 12
~# QEMU: Terminated
```

问题回顾

回顾实验课教案与代码，以下问题可以帮助大家梳理从开始到现在，操作系统的整体运作方式，加深对于操作系统的印象：

1. 为什么系统里面大量分布着`Arc`，而不是使用Rust的引用来进行资源回收？`Arc`与Rust引用的区别是什么？
2. 第一个用户态进程是怎么启动的？在启动之前OSTD、Kernel、Init thread分别做了什么？
3. 初始化用户态进程的过程除了解析ELF文件，还做了什么额外的操作？为什么需要这些操作？
4. 标准库是什么？一定指的是Linux上的C语言标准库吗？为什么使用C语言标准库编写的程序可以在我们写的内核中执行？想想OSTD名字的全称与原因