OS-lab5 report

刘泓尊 2018011446 计84

Chapter5 实现的内容

1. 增加了进程抽象，提供了进程相关的系统调用: `getpid` , `fork` , `exec` , `waitpid` , `sapwn`
2. 用户程序封装了新的系统调用，实现了用户程序的shell，增加可交互性
3. 实现了初始进程的创建、进程调度与切换、进程加载与生成、进程资源回收等功能，支持父子进程
4. 实现处理器监视器，提供idle进程便于执行流切换。对用户进程屏蔽了调度执行流
5. 解决了实现过程中的很多死锁问题, 支持 `waitpid` 的阻塞和非阻塞方式

编程作业

waitpid系统调用

之前作业要求是实现阻塞的，我就实现了，没想到后来又改成非阻塞了。这里简要叙述一下阻塞的实现，位于 `process.rs` 的 `sys_waitpid_blocking` .

首先获得当前进程PCB，判断有没有子进程，如果没有子进程返回-1. 存在子进程的话，就进入一个loop, 如果有子进程退出就返回该pid, 不然就调用 `suspend_current_and_run_next` 切换进程。相当于这时是在其他进程和内核进程之间来回切换，直到内核发现有子进程退出。一个坑点是已经要及时释放锁！！！ 在 `suspend_current_and_run_next` 之前要释放当前进程锁，不然 `suspend_current_and_run_next` 可能又会请求这个锁。死锁问题是我lab5解决时间最久的问题。

spawn 系统调用

类似fork + exec. 首先要根据当前进程页表把file* 参数翻译成物理地址，调用 `translated_str` 即可。根据名字获得程序数据data之后，在当前进程fork处子进程task，然后调用task.exec(data)来加载数据。最后向调度队列添加这个task即可。实现大都是参考fork 和exec的实现。

initproc的退出

在tutorial 框架下，把initproc设置成usertest会导致OS无法结束，这是因为在 `exit_current_and_run_next` 在结束 `INITPROC` 的时候存在死锁。当只剩一个进程initproc的时候，OS会在这里两次获得initproc的内部锁，导致程序无法退出。一个简单的解决办法就是判断一下当前退出进程和 `initproc` 是否相同。相同就不重复获得锁了。

```rust
pub fn exit_current_and_run_next(exit_code: i32) {
    // take from Processor
    // 将当前进程控制块从处理器监控 PROCESSOR 中取出而不是得到一份拷贝
    // 为了正确维护进程控制块的引用计数
    let task = take_current_task().unwrap();
    // **** hold current PCB lock
    let mut inner = task.acquire_inner_lock();
    // Change status to Zombie
    inner.task_status = TaskStatus::Zombie;
    // Record exit code
    // 将传入的退出码 exit_code 写入进程控制块中，后续父进程在 waitpid 的时候可以收集
    inner.exit_code = exit_code;
    // do not move to its parent but under initproc

    // ++++++ hold initproc PCB lock here
    // 将当前进程的所有子进程挂在初始进程 initproc 下面
    if task.getpid() != INITPROC.getpid() {
        let mut initproc_inner = INITPROC.acquire_inner_lock();
        for child in inner.children.iter() { // 遍历每个子进程
            child.acquire_inner_lock().parent = Some(Arc::downgrade(&INITPROC)); // 修改其父进程为初始进程
            initproc_inner.children.push(child.clone()); // 加入初始进程的孩子向量中
        }
    }
    // ++++++ release parent PCB lock here      liuhz, 3 weeks ago • add lab5 tutorial codes
```

测试结果截图

直接把 `INITPROC` 设置成了 `initproc` (由 `usertest` 复制而来)，下面是运行结果：

```
Test getpid OK! pid = Test getpid OK! pid = 41
40
Test spawn0 OK!
Usertests: Test ch5_spawn0 in Process 1 exited with code 0
Usertests: Running ch5_spawn1
new child 40
Test wait OK!
Test waitpid OK!
Usertests: Test ch5_spawn1 in Process 1 exited with code 0
ch5 Usertests passed!
```

问答作业

1. **fork + exec 的一个比较大的问题是 fork 之后的内存页/文件等资源完全没有使用就废弃了，针对这一点，有什么改进策略?**

   可以在fork前设置一些参数改变fork的默认行为。比如打开文件的时候，在open()系统调用中标记“close on fork”, 那么fork的时候这个文件描述符就不会泄露给子进程；在内存映射之后标记某段内存 “dont fork”，使得fork的时候这段内存映射不会复制给子进程。

   在Linux中可以通过 `madvise()` 设置 `MADV_DONTFORK` 参数，不继承某段内存映射；可以通过 `open` 时设置 `O_CLOEXEC` , 或在 `fcntl()` 中设置 `FD_CLOEXEC` , 来在exec()的时候关闭一些文件描述符。

   Unix也有 `vfork()` 作为改进，这种fork并不使用内存页的写时复制(COW), 而是和父进程共享。不消耗额外的内存，和 `exec()` 搭配的时候速度比较快。但是 `vfork` 和 `exec` 中间就不方便执行一些操作了，因为这中间的任何修改都对父进程可见。

2. **其实使用了题(1)的策略之后，fork + exec 所带来的无效资源的问题已经基本被解决了，但是今年来 fork 还是在被不断的批判，那么到底是什么正在"杀死"fork? 可以参考 论文**

   参考了论文中对fork的评价，总结了以下几点缺陷：

   - 慢且实现复杂。随着进程的语义越来越复杂，可能持有了较大的内存映射、很多锁、文件描述符等，在fork的时候需要一一继承，带来了很大的时间空间开销；而且在文档中必须

详细描述每一种资源在fork时的行为，增大了应用成本。甚至fork前还需要flush一些IO buffer, 确保buffer里的内容不被继承，这增加了用户程序的成本。
    - 破坏了隔离性和安全性。fork出的进程 和 父进程有相同的文件描述符、内存映射等，共享很多资源，违背了最小权限原则；同时也破坏了地址空间的布局随机性。
    - 不是线程安全的。如果父进程正在持有锁，fork子进程后企图获得这个锁，就会发生死锁。比如线程1 `malloc()` 的时候享有heap lock，在这时线程2调用fork并malloc, 死锁就会发生。
    - 在异构硬件上不兼容。fork更适用于单核、单个地址空间的情景。而对于现在的GPU，有内核旁路的NIC等硬件，OS无法复制这些硬件上的状态。
3. **fork 当年被设计并称道肯定是有其好处的。请使用 带初始参数 的 spawn 重写如下 fork 程序，然后描述 fork 有那些好处。注意:使用"伪代码"传达意思即可，spawn 接口可以自定义。可以写多个文件。**

```
fn main() {
    let a = get_a();
    if fork() == 0 { // child process
        let b = get_b();
        println!("a + b = {}", a + b);
        exit(0);
    }
    println!("a = {}", a);
    0
}
```

spwan接口：`fn spawn(file: *const u8) -> isize;` 成功返回子进程pid, 失败返回-1

spwan重写后的程序：

```
// proc_a.rs
fn main() {
    let a = get_a();
    cpid = spawn("proc_b");
    if cpid < 0 {
        println!("invalid file name");
    }
    // 如果要等待子进程结束，就使用下面注释掉的代码
    //loop {
    //    let mut exit_code: i32 = 0;
    //    let pid = wait(&mut exit_code);
    //    if pid == -1 {
    //        yield_();
    //        continue;
    //    } else {
    //        break;
    //    }
    //}
    println!("a = {}", a);
    0
}

```

```rust
// proc_b.rs
fn main() {
    let a = get_a();
    let b = get_b();
    println!("a + b = {}", a + b);
    exit(0);
}
```

fork灵活度较高，可以看到fork可以很大程度上实现代码重用，fork可以派生多个程序副本，从同一个二进制文件运行不同的函数，同时还可以继承一些变量，免于进程间通信。fork没有参数，在某些场景下比较简洁。

而spawn需要路径参数，这就要求需要多个二进制文件相互配合来完成某个并发功能，当b进程需要a进程的某个数据的时候，还需要进程间通信。

4. **描述进程执行的几种状态，以及 fork/exec/wait/exit 对于状态的影响。**

   在我实现的OS中，进程状态包括 **就绪**(Ready，可以被处理机调度), **运行**(Running，当前正在运行的进程), **退出**(Zombie，程序已经结束)3种状态。

   理论课上讲的还有创建(Create)状态，表示正在被创建，还没有就绪，但实际上在我简单的OS上并不需要；理论课上的等待(Waiting)状态也可以用系统调用wait(waitpid)实现。所以我们的OS用3个状态就可以了。

   `fork` : 对于调用fork的父进程，fork子进程后返回继续运行，所以 **父进程 从 运行(Running)->运行(Running)。子进程**则被创建出来，状态是 **就绪(Ready)**.

   `exec` : exec一般由fork出来的进程调用，会重新加载elf数据，继续运行新的二进制文件。所以状态是 **运行(Running)->运行(Running)。**

   `wait` : 如果wait的阻塞在**内核**实现，逻辑就是查看子进程状态->子进程没结束就调度到其他进程->结束了就直接返回。所以调用wait的程序要么会 从 **运行(Running)->运行(Running)**，要么会 从 **运行(Running)->就绪(Ready)**.

   如果wait的阻塞在**用户**库实现，那么无论进程有没有结束，都会返回用户程序，之后调用yield与否和wait是解耦的。所以状态是 **运行(Running)->运行(Running)。**

   `exit` : 调用exit, 程序会从 **运行(Running)->退出(Zombie)**. 之后OS会执行任务切换。