chp5

改动的代码文件和注释

5.1

`wait` 用于等待任意一个子进程， `waitpid` 用于等待特定子进程

```rust
/// 等待任意一个子进程
pub fn wait(exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(-1, exit_code as *mut _) {
            -2 => {
                //等待的子进程均未结束则返回 -2;
                sys_yield();
            }
            n => {
                //否则返回结束的子进程的进程 ID
                return n;
            }
        }
    }
}

pub fn waitpid(pid: usize, exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(pid as isize, exit_code as *mut _) {
            -2 => {
                sys_yield();
            }
            n => {
                return n;
            }
        }
    }
}
```

`shell` 程序的执行

```rust
pub fn main() -> i32 {
    println!("Rust user shell");
    let mut line: String = String::new();
    print!(">> ");
    flush();
    loop {
        let c = getchar();
        match c {
            //回车
            LF | CR => {
                print!("\n");
                if !line.is_empty() {
                    line.push('\0');
                    let pid = fork();
                    //子进程执行
                    if pid == 0 {
                        // child process
                        if exec(line.as_str(), &[0 as *const u8]) == -1 {
                            //返回值为 -1 , 说明在应用管理器中找不到对应名字的应用
                            println!("Error when executing!");
                            return -4;
                        }
                        unreachable!();
                    } else {
                        let mut exit_code: i32 = 0;
                        let exit_pid = waitpid(pid as usize, &mut exit_code);
                        assert_eq!(pid, exit_pid);
                        println!("Shell: Process {} exited with code {}", pid, exit_code);
                    }
                    line.clear();
                }
                print!(">> ");
                flush();
            }
            //退格 backspace 替换字符为空格
            BS | DL => {
                if !line.is_empty() {
                    print!("{}", BS as char);
                    print!(" ");
                    print!("{}", BS as char);
                    flush();
                    line.pop();
                }
```

```
            }
            _ => {
                //其他字符加入进line print到屏幕上
                print!("{}", c as char);
                flush();
                line.push(c as char);
            }
        }
    }
}
```

5.2

`loader.rs` 中，我们用一个全局可见的 *只读* 向量 `APP_NAMES` 来按照顺序将所有应用的名字保存在内存中

```
#[allow(unused)]
///get app data from name
pub fn get_app_data_by_name(name: &str) -> Option<&'static [u8]> {
    // 三查找获得应用的 ELF 数据
    let num_app = get_num_app();
    (0..num_app)
        .find(|&i| APP_NAMES[i] == name)
        .map(get_app_data)
}
///list all apps
pub fn list_apps() {
    // 打印出所有可用应用的名字
    println!("/**** APPS ****");
    for app in APP_NAMES.iter() {
        println!("{}", app);
    }
    println!("**************/");
}
```

进程标识符 `PidAllocator`

```rust
pub struct RecycleAllocator {
    current: usize,
    recycled: Vec<usize>,
}

impl RecycleAllocator {
    // 初始化
    pub fn new() -> Self {
        RecycleAllocator {
            current: 0,
            recycled: Vec::new(),
        }
    }
    // 分配pid
    pub fn alloc(&mut self) -> usize {
        // 如果有 使用回收的
        if let Some(id) = self.recycled.pop() {
            id
        } else {
            // 重新分配
            self.current += 1;
            self.current - 1
        }
    }
    // 回收pid
    pub fn dealloc(&mut self, id: usize) {
        // 首先断言 PID 必须是已分配的（小于 current）
        assert!(id < self.current);
        // 断言 PID 没有被重复回收
        assert!(!self.recycled.iter().any(|i| *i == id), "id {} has been deallocated!", id);
        self.recycled.push(id);
    }
}
```

全局分配进程标识符的接口 `pid_alloc`

```rust
/// Allocate a new PID
pub fn pid_alloc() -> PidHandle {
    // 全局函数调用alloc 分配pid
    PidHandle(PID_ALLOCATOR.exclusive_access().alloc())
}
```

资源回收

```rust
impl Drop for PidHandle {
    //自动资源回收
    fn drop(&mut self) {
        //println!("drop pid {}", self.0);
        PID_ALLOCATOR.exclusive_access().dealloc(self.0);
    }
}
```

内核栈 `KernelStack`

在内核栈 `KernelStack` 中保存着它所属进程的 PID：

```rust
// os/src/task/pid.rs

pub struct KernelStack {
    pid: usize,
}
```

内核栈位置计算 `kernel_stack_position`

```rust
/// Return (bottom, top) of a kernel stack in kernel space.
/// 内核栈位置计算
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    /*
    TRAMPOLINE 存放跳板代码 0xFFFFFFFFFFFFF000
    每个内核栈占用 KERNEL_STACK_SIZE + PAGE_SIZE 的空间
    Stack向下增长 top - KERNEL_STACK_SIZE
    */
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);

    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
    // 返回(bottom, top)，表示内核栈的地址范围
}
```

KernelStack 的实现

```rust
impl KernelStack {
    /// new一个KernelStack
    pub fn new(pid_handle: &PidHandle) -> Self {
        let pid = pid_handle.0;
        let (kernel_stack_bottom, kernel_stack_top) = kernel_stack_position(pid);
        KERNEL_SPACE.exclusive_access().insert_framed_area(
            // 将 [kernel_stack_bottom, kernel_stack_top) 映射到物理内存
            kernel_stack_bottom.into(),
            kernel_stack_top.into(),
            MapPermission::R | MapPermission::W
        );
        KernelStack { pid: pid_handle.0 }
    }
    /// Push a variable of type T into the top of the KernelStack and return its raw pointer
    #[allow(unused)]
    pub fn push_on_top<T>(&self, value: T) -> *mut T where T: Sized {
        let kernel_stack_top = self.get_top(); //获取当前栈顶地址
        let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as *mut T; //计算value 的存
        unsafe {
            *ptr_mut = value; //写入数据
        }
        ptr_mut
    }
    /// Get the top of the KernelStack
    pub fn get_top(&self) -> usize {
        let (_, kernel_stack_top) = kernel_stack_position(self.pid);
        kernel_stack_top
    }
}
```

进程控制块 TaskControlBlock

`TaskControlBlockInner` 提供的方法

```rust
impl TaskControlBlockInner {
    /// get the trap context
    pub fn get_trap_cx(&self) -> &'static mut TrapContext {
        //返回 陷阱上下文(TrapContext)的可变引用
        self.trap_cx_ppn.get_mut()
    }
    /// get the user token
    pub fn get_user_token(&self) -> usize {
        self.memory_set.token()
    }
    fn get_status(&self) -> TaskStatus {
        self.task_status
    }
    pub fn is_zombie(&self) -> bool {
        // 检查当前任务是否是 僵尸状态
        self.get_status() == TaskStatus::Zombie
    }
}
```

`TaskControlBlock` 提供的方法

```rust
impl TaskControlBlock {
    pub fn inner_exclusive_access(&self) -> RefMut<'_, TaskControlBlockInner> {
        self.inner.exclusive_access()
    }
    pub fn getpid(&self) -> usize {
        self.pid.0
    }
    //以下三个没有实现
    pub fn new(elf_data: &[u8]) -> Self {...}
    pub fn exec(&self, elf_data: &[u8]) {...}
    pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {...}
}
```

任务管理器 `TaskManager`

任务管理器的结构:双端队列

和方法:

```rust
/// A simple FIFO scheduler.
impl TaskManager {
    ///Creat an empty TaskManager
    pub fn new() -> Self {
        Self {
            ready_queue: VecDeque::new(),
        }
    }
    /// Add process back to ready queue
    pub fn add(&mut self, task: Arc<TaskControlBlock>) {
        self.ready_queue.push_back(task);
    }
    /// Take a process out of the ready queue
    /// 取出下一个任务
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.ready_queue.pop_front()
    }
}

lazy_static! {
    /// TASK_MANAGER instance through lazy_static!
    pub static ref TASK_MANAGER: UPSafeCell<TaskManager> = unsafe {
        UPSafeCell::new(TaskManager::new())
    };
}

/// Add process to ready queue
pub fn add_task(task: Arc<TaskControlBlock>) {
    //trace!("kernel: TaskManager::add_task");
    //首先独占 然后添加task
    TASK_MANAGER.exclusive_access().add(task);
}

/// Take a process out of the ready queue
pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
    //trace!("kernel: TaskManager::fetch_task");
    //取出任务
    TASK_MANAGER.exclusive_access().fetch()
}
```

处理器管理结构

处理器管理结构 `Processor` 负责维护从任务管理器 `TaskManager` 分离出去的那部分 CPU 状态：

```
// os/src/task/processor.rs

pub struct Processor {
    ///The task currently executing on the current processor
    current: Option<Arc<TaskControlBlock>>, //当前正在 CPU 上运行的任务

    ///The basic control flow of each core, helping to select and switch process
    /// idle_task_cx是一个 任务上下文(TaskContext),它保存了 调度器(Scheduler)自身的寄存器状态
    /// //表示当前处理器上的 idle 控制流的任务上下文的地址
    idle_task_cx: TaskContext,
}
```

方法

```rust
impl Processor {
    ///Get current task in moving semanteme
    pub fn take_current(&mut self) -> Option<Arc<TaskControlBlock>> {
        //取出当前任务,取出后 current 变为 None
        self.current.take()
    }

    ///Get current task in cloning semanteme
    pub fn current(&self) -> Option<Arc<TaskControlBlock>> {
        //获取当前任务(克隆语义)
        self.current.as_ref().map(Arc::clone)
    }
}

/// Get current task through take, leaving a None in its place
pub fn take_current_task() -> Option<Arc<TaskControlBlock>> {
    PROCESSOR.exclusive_access().take_current()
}

/// Get a copy of the current task
pub fn current_task() -> Option<Arc<TaskControlBlock>> {
    PROCESSOR.exclusive_access().current()
}

/// Get the current user token(addr of page table)
pub fn current_user_token() -> usize {
    let task = current_task().unwrap();
    task.get_user_token()
}

///Get the mutable reference to trap context of current task
pub fn current_trap_cx() -> &'static mut TrapContext {
    // 获取当前trap上下文
    current_task().unwrap().inner_exclusive_access().get_trap_cx()
}

///Return to idle control flow for new scheduling
```

任务调度的 idle 控制流

idle 控制流，运行在每个核各自的启动栈上,从任务管理器中选一个任务在当前的 core 上面执行

持续运行任务

```
// os/src/task/processor.rs

impl Processor {
     ///Get mutable reference to `idle_task_cx`
    /// 获取空闲上下文指针
    fn get_idle_task_cx_ptr(&mut self) -> *mut TaskContext {
        &mut self.idle_task_cx as *mut _
    }
 }

 ///The main part of process execution and scheduling
///Loop `fetch_task` to get the process that needs to run, and switch the process through `__sw:
 pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        // 从就绪队列获取任务
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            // access coming task TCB exclusively
            let mut task_inner = task.inner_exclusive_access();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running; //设置TaskStatus为Running
            // release coming task_inner manually
            drop(task_inner); // 手动释放锁
            // release coming task TCB manually
            processor.current = Some(task); //更新任务
            // release processor manually
            drop(processor);
            unsafe {
                __switch(idle_task_cx_ptr, next_task_cx_ptr); //切换任务
            }
        } else {
            warn!("no tasks available in run_tasks");
        }
    }
}
```

`schedule` 函数来切换到 idle 控制流并开启新一轮的任务调度

```rust
///Return to idle control flow for new scheduling
pub fn schedule(switched_task_cx_ptr: *mut TaskContext) {
    //当前任务主动放弃 CPU(如调用 yield 或阻塞).

    let mut processor = PROCESSOR.exclusive_access();
    let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
    drop(processor);
    unsafe {
        //切换回 idle_task_cx,重新进入调度循环.
        __switch(switched_task_cx_ptr, idle_task_cx_ptr);
    }
}
```