|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 北京交通大学软件学院  **《操作系统》课程**  **实验报告** | | |

|  |
| --- |
| **姓名:韩熔** |
| **学号:20301036** |

目录

[1 实验目的 3](#_Toc118133029)

[2 实验过程设计 3](#_Toc118133030)

[3 源代码及注释 3](#_Toc118133031)

[4 运行结果与分析 4](#_Toc118133032)

[5 实验总结 4](#_Toc118133033)

# 1 实验目的

实现进程及进程的管理，让开发者能够控制程序的运行 。

开发一个用户终端程序或称命令行应用，形成用户与操作系统进行交互的命令行界面。

# 2 实验过程设计

学习进程知识，建立进程模型与重要系统调用

重新设计一些进程管理的核心数据结构包含的内容及接口

进程管理机制的设计与实现

# 3 源代码及注释

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/process.rs |
| **主要功能:** | 实现系统调用 |
| **源代码:**  **//fork 系统调用,其他所有的进程都是通过一个名为 fork 的系统调用来创建的。**  **pub fn sys\_fork() -> isize {**  **let current\_task = current\_task().unwrap();**  **let new\_task = current\_task.fork();**  **let new\_pid = new\_task.pid.0;**  **// modify trap context of new\_task, because it returns immediately after switching**  **let trap\_cx = new\_task.inner\_exclusive\_access().get\_trap\_cx();**  **// we do not have to move to next instruction since we have done it before**  **// for child process, fork returns 0**  **trap\_cx.x[10] = 0;**  **// add new task to scheduler**  **add\_task(new\_task);**  **new\_pid as isize**  **}**  **//exec 系统调用**  **/// 功能：将当前进程的地址空间清空并加载一个特定的可执行文件，返回用户态后开始它的执行。**  **/// 参数：path 给出了要加载的可执行文件的名字；**  **/// 返回值：如果出错的话（如找不到名字相符的可执行文件）则返回 -1，否则不应该返回。**  **/// syscall ID：221**  **pub fn sys\_exec(path: \*const u8) -> isize {**  **let token = current\_user\_token();**  **let path = translated\_str(token, path);**  **if let Some(data) = get\_app\_data\_by\_name(path.as\_str()) {**  **let task = current\_task().unwrap();**  **task.exec(data);**  **0**  **} else {**  **-1**  **}**  **}**  **//waitpid 系统调用**  **/// 功能：当前进程等待一个子进程变为僵尸进程，回收其全部资源并收集其返回值。**  **/// 参数：pid 表示要等待的子进程的进程 ID，如果为 -1 的话表示等待任意一个子进程；**  **/// exit\_code 表示保存子进程返回值的地址，如果这个地址为 0 的话表示不必保存。**  **/// 返回值：如果要等待的子进程不存在则返回 -1；否则如果要等待的子进程均未结束则返回 -2；**  **/// 否则返回结束的子进程的进程 ID。**  **/// syscall ID：260**  **pub fn sys\_waitpid(pid: isize, exit\_code\_ptr: \*mut i32) -> isize {**  **let task = current\_task().unwrap();**  **// find a child process**  **// ---- access current TCB exclusively**  **let mut inner = task.inner\_exclusive\_access();**  **if !inner**  **.children**  **.iter()**  **.any(|p| pid == -1 || pid as usize == p.getpid())**  **{**  **return -1;**  **// ---- release current PCB**  **}**  **let pair = inner.children.iter().enumerate().find(|(\_, p)| {**  **// ++++ temporarily access child PCB lock exclusively**  **p.inner\_exclusive\_access().is\_zombie() && (pid == -1 || pid as usize == p.getpid())**  **// ++++ release child PCB**  **});**  **if let Some((idx, \_)) = pair {**  **let child = inner.children.remove(idx);**  **// confirm that child will be deallocated after removing from children list**  **assert\_eq!(Arc::strong\_count(&child), 1);**  **let found\_pid = child.getpid();**  **// ++++ temporarily access child TCB exclusively**  **let exit\_code = child.inner\_exclusive\_access().exit\_code;**  **// ++++ release child PCB**  **\*translated\_refmut(inner.memory\_set.token(), exit\_code\_ptr) = exit\_code;**  **found\_pid as isize**  **} else {**  **-2**  **}**  **// ---- release current PCB lock automatically**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | user/src/lib.rs |
| **主要功能:** | 封装系统调用 |
| **源代码:**  **//表示等待任意一个子进程结束，根据 sys\_waitpid 的约定它需要传的 pid 参数为 -1**  **pub fn wait(exit\_code: &mut i32) -> isize {**  **loop {**  **match sys\_waitpid(-1, exit\_code as \*mut \_) {**  **-2 => { yield\_(); }**  **// -1 or a real pid**  **exit\_pid => return exit\_pid,**  **}**  **}**  **}**  **//waitpid 则等待一个进程标识符的值为pid 的子进程结束**  **pub fn waitpid(pid: usize, exit\_code: &mut i32) -> isize {**  **loop {**  **match sys\_waitpid(pid as isize, exit\_code as \*mut \_) {**  **-2 => { yield\_(); }**  **// -1 or a real pid**  **exit\_pid => return exit\_pid,**  **}**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | user/src/bin/initproc.rs |
| **主要功能:** | 用户初始程序 |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  **#[macro\_use]**  **extern crate user\_lib;**  **use user\_lib::{exec, fork, wait, yield\_};**  **#[no\_mangle]**  **fn main() -> i32 {**  **if fork() == 0 {**  **//表示子进程，通过 exec 执行 shell 程序 user\_shell**  **exec("user\_shell\0");**  **} else {**  **//调用 fork 的用户初始程序 initproc 自身**  **//不断循环调用 wait 来等待那些被移交到它下面的子进程并回收它们占据的资源**  **loop {**  **let mut exit\_code: i32 = 0;**  **let pid = wait(&mut exit\_code);**  **if pid == -1 {**  **yield\_();**  **continue;**  **}**  **println!(**  **"[initproc] Released a zombie process, pid={}, exit\_code={}",**  **pid, exit\_code,**  **);**  **}**  **}**  **0**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/syscall/fs.rs |
| **主要功能:** | 文件和文件系统相关的系统调用 |
| **源代码:**  **//! File and filesystem-related syscalls**  **use crate::mm::translated\_byte\_buffer;**  **use crate::sbi::console\_getchar;**  **use crate::task::{current\_user\_token, suspend\_current\_and\_run\_next};**  **const FD\_STDIN: usize = 0;**  **const FD\_STDOUT: usize = 1;**  **pub fn sys\_write(fd: usize, buf: \*const u8, len: usize) -> isize {**  **match fd {**  **FD\_STDOUT => {**  **let buffers = translated\_byte\_buffer(current\_user\_token(), buf, len);**  **for buffer in buffers {**  **print!("{}", core::str::from\_utf8(buffer).unwrap());**  **}**  **len as isize**  **}**  **\_ => {**  **panic!("Unsupported fd in sys\_write!");**  **}**  **}**  **}**  **/// 功能：从文件中读取一段内容到缓冲区。**  **/// 参数：fd 是待读取文件的文件描述符，切片 buffer 则给出缓冲区。**  **/// 返回值：如果出现了错误则返回 -1，否则返回实际读到的字节数。**  **/// syscall ID：63**  **pub fn sys\_read(fd: usize, buf: \*const u8, len: usize) -> isize {**  **match fd {**  **FD\_STDIN => {**  **assert\_eq!(len, 1, "Only support len = 1 in sys\_read!");**  **let mut c: usize;**  **loop {**  **c = console\_getchar();**  **if c == 0 {**  **suspend\_current\_and\_run\_next();**  **continue;**  **} else {**  **break;**  **}**  **}**  **let ch = c as u8;**  **let mut buffers = translated\_byte\_buffer(current\_user\_token(), buf, len);**  **unsafe {**  **buffers[0].as\_mut\_ptr().write\_volatile(ch);**  **}**  **1**  **}**  **\_ => {**  **panic!("Unsupported fd in sys\_read!");**  **}**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | user/src/bin/user\_shell.rs |
| **主要功能:** | 实现shell 程序 user\_shell |
| **源代码:**  **1 // user/src/bin/user\_shell.rs**  **2**  **3 #![no\_std]**  **4 #![no\_main]**  **5**  **6 extern crate alloc;**  **7**  **8 #[macro\_use]**  **9 extern crate user\_lib;**  **10**  **11 const LF: u8 = 0x0au8;**  **12 const CR: u8 = 0x0du8;**  **13 const DL: u8 = 0x7fu8;**  **14 const BS: u8 = 0x08u8;**  **15**  **16 use alloc::string::String;**  **17 use user\_lib::{fork, exec, waitpid, yield\_};**  **18 use user\_lib::console::getchar;**  **19**  **20 #[no\_mangle]**  **21 pub fn main() -> i32 {**  **22 println!("Rust user shell");**  **23 let mut line: String = String::new();**  **24 print!(">> ");**  **25 loop {**  **26 let c = getchar();**  **27 match c {**  **28 LF | CR => {**  **29 println!("");**  **30 if !line.is\_empty() {**  **31 line.push('\0');**  **32 let pid = fork();**  **33 if pid == 0 {**  **34 // user\_shell 会 fork 出一个子进程并试图通过 exec 系统调用执行一个应用，应用的名字在字符串 line 中给出**  **//对 exec 的返回值进行了判断，如果返回值为 -1 则说明在应用管理器中找不到名字相同的应用，此时子进程就直接打印错误信息并退出；反之 exec 则根本不会返回，而是开始执行目标应用。**  **35 if exec(line.as\_str()) == -1 {**  **36 println!("Error when executing!");**  **37 return -4;**  **38 }**  **39 unreachable!();**  **40 } else {**  **41 let mut exit\_code: i32 = 0;**  **42 let exit\_pid = waitpid(pid as usize, &mut exit\_code);**  **43 assert\_eq!(pid, exit\_pid);**  **44 println!(**  **45 "Shell: Process {} exited with code {}",**  **46 pid, exit\_code**  **47 );**  **48 }**  **49 line.clear();**  **50 }**  **51 print!(">> ");**  **52 }**  **//如果用户输入退格键，首先我们需要将屏幕上当前行的最后一个字符用空格替换掉，这可以通过输入一个特殊的退格字节 BS 来实现。其次，user\_shell 进程内维护的 line 也需要弹出最后一个字符。**  **53 BS | DL => {**  **54 if !line.is\_empty() {**  **55 print!("{}", BS as char);**  **56 print!(" ");**  **57 print!("{}", BS as char);**  **58 line.pop();**  **59 }**  **60 }**  **//如果用户输入了一个其他字符，它将会被视为用户的正常输入，我们直接将它打印在屏幕上并加入到 line 中**  **61 \_ => {**  **62 print!("{}", c as char);**  **63 line.push(c as char);**  **64 }**  **65 }**  **66 }**  **67 }** | |

|  |  |
| --- | --- |
| **文件名:** | os/build.rs |
| **主要功能:** | 读取位于 user/src/bin 中应用程序对应的执行文件，并生成 link\_app.S |
| **源代码:**  **1 // os/build.rs**  **2**  **3 for i in 0..apps.len() {**  **4 writeln!(f, r#" .quad app\_{}\_start"#, i)?;**  **5 }**  **6 writeln!(f, r#" .quad app\_{}\_end"#, apps.len() - 1)?;**  **7//按照顺序将各个应用的名字通过 .string 伪指令放到数据段中**  **8 writeln!(f, r#"**  **9 .global \_app\_names**  **10 \_app\_names:"#)?;**  **11 for app in apps.iter() {**  **12 writeln!(f, r#" .string "{}""#, app)?;**  **13 }**  **14**  **15 for (idx, app) in apps.iter().enumerate() {**  **16 ...**  **17 }** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/loader.rs |
| **主要功能:** | 加载器 loader.rs |
| **源代码:**  **//分析 link\_app.S 中的内容，并用一个全局可见的 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**  **lazy\_static! {**  **static ref APP\_NAMES: Vec<&'static str> = {**  **let num\_app = get\_num\_app();**  **extern "C" { fn \_app\_names(); }**  **let mut start = \_app\_names as usize as \*const u8;**  **let mut v = Vec::new();**  **unsafe {**  **for \_ in 0..num\_app {**  **let mut end = start;**  **while end.read\_volatile() != '\0' as u8 {**  **end = end.add(1);**  **}**  **let slice = core::slice::from\_raw\_parts(start, end as usize - start as usize);**  **let str = core::str::from\_utf8(slice).unwrap();**  **v.push(str);**  **start = end.add(1);**  **}**  **}**  **v**  **};**  **}**  **//按照应用的名字来查找获得应用的 ELF 数据**  **pub fn get\_app\_data\_by\_name(name: &str) -> Option<&'static [u8]> {**  **let num\_app = get\_num\_app();**  **(0..num\_app)**  **.find(|&i| APP\_NAMES[i] == name)**  **.map(|i| get\_app\_data(i))**  **}**  **//在内核初始化时被调用，可以打印出所有可用的应用的名字**  **pub fn list\_apps() {**  **println!("/\*\*\*\* APPs \*\*\*\*");**  **for app in APP\_NAMES.iter() {**  **println!("{}", app);**  **}**  **println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*/")**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/pid.rs |
| **主要功能:** | 实现PidAllocator |
| **源代码:**  **struct PidAllocator {**  **current: usize,**  **recycled: Vec<usize>,**  **}**  **//使用简单栈式分配策略的进程标识符分配器 PidAllocator ，并将其全局实例化为 PID\_ALLOCATOR**  **impl PidAllocator {**  **pub fn new() -> Self {**  **PidAllocator {**  **current: 0,**  **recycled: Vec::new(),**  **}**  **}**  **pub fn alloc(&mut self) -> PidHandle {**  **if let Some(pid) = self.recycled.pop() {**  **PidHandle(pid)**  **} else {**  **self.current += 1;**  **PidHandle(self.current - 1)**  **}**  **}**  **pub fn dealloc(&mut self, pid: usize) {**  **assert!(pid < self.current);**  **assert!(**  **self.recycled.iter().find(|ppid| \*\*ppid == pid).is\_none(),**  **"pid {} has been deallocated!", pid**  **);**  **self.recycled.push(pid);**  **}**  **}**  **lazy\_static! {**  **static ref PID\_ALLOCATOR : UPSafeCell<PidAllocator> = unsafe {**  **UPSafeCell::new(PidAllocator::new())**  **};**  **}**  **//PidAllocator::alloc 将会分配出去一个将 usize 包装之后的 PidHandle 。我们将其包装为一个全局分配进程标识符的接口 pid\_alloc 提供给内核的其他子模块**  **pub fn pid\_alloc() -> PidHandle {**  **PID\_ALLOCATOR.exclusive\_access().alloc()**  **}**  **//为 PidHandle 实现 Drop Trait 来允许编译器进行自动的资源回收**  **impl Drop for PidHandle {**  **fn drop(&mut self) {**  **PID\_ALLOCATOR.exclusive\_access().dealloc(self.0);**  **}**  **}**  **//在内核栈 KernelStack 中保存着它所属进程的 PID**  **pub struct KernelStack {**  **pid: usize,**  **}**  **/// Return (bottom, top) of a kernel stack in kernel space.**  **pub fn kernel\_stack\_position(app\_id: usize) -> (usize, usize) {**  **let top = TRAMPOLINE - app\_id \* (KERNEL\_STACK\_SIZE + PAGE\_SIZE);**  **let bottom = top - KERNEL\_STACK\_SIZE;**  **(bottom, top)**  **}**  **impl KernelStack {**  **///Create a kernelstack from pid**  **// new 方法可以从一个 PidHandle ，也就是一个已分配的进程标识符中对应生成一个内核栈 KernelStack**  **//调用了kernel\_stack\_position 函数来根据进程标识符计算内核栈在内核地址空间中的位置**  **pub fn new(pid\_handle: &PidHandle) -> Self {**  **let pid = pid\_handle.0;**  **let (kernel\_stack\_bottom, kernel\_stack\_top) = kernel\_stack\_position(pid);**  **//将一个逻辑段插入内核地址空间 KERNEL\_SPACE 中**  **KERNEL\_SPACE.exclusive\_access().insert\_framed\_area(**  **kernel\_stack\_bottom.into(),**  **kernel\_stack\_top.into(),**  **MapPermission::R | MapPermission::W,**  **);**  **KernelStack { pid: pid\_handle.0 }**  **}**  **#[allow(unused)]**  **///Push a value on top of kernelstack**  **//将一个类型为 T 的变量压入内核栈顶并返回其裸指针**  **//用到了get\_top 方法来获取当前内核栈顶在内核地址空间中的地址**  **pub fn push\_on\_top<T>(&self, value: T) -> \*mut T**  **where**  **T: Sized,**  **{**  **let kernel\_stack\_top = self.get\_top();**  **let ptr\_mut = (kernel\_stack\_top - core::mem::size\_of::<T>()) as \*mut T;**  **unsafe {**  **\*ptr\_mut = value;**  **}**  **ptr\_mut**  **}**  **///Get the value on the top of kernelstack**  **pub fn get\_top(&self) -> usize {**  **let (\_, kernel\_stack\_top) = kernel\_stack\_position(self.pid);**  **kernel\_stack\_top**  **}**  **}**  **//内核栈 KernelStack 也用到了 RAII 的思想，具体来说，实际保存它的物理页帧的生命周期与它绑定在一起，当 KernelStack 生命周期结束后，这些物理页帧也将会被编译器自动回收：**  **impl Drop for KernelStack {**  **fn drop(&mut self) {**  **let (kernel\_stack\_bottom, \_) = kernel\_stack\_position(self.pid);**  **let kernel\_stack\_bottom\_va: VirtAddr = kernel\_stack\_bottom.into();**  **KERNEL\_SPACE**  **.exclusive\_access()**  **.remove\_area\_with\_start\_vpn(kernel\_stack\_bottom\_va.into());**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/task.rs |
| **主要功能:** | 进程控制块 |
| **源代码:**  **pub struct TaskControlBlock {**  **// immutable**  **pub pid: PidHandle,**  **pub kernel\_stack: KernelStack,**  **// mutable**  **inner: UPSafeCell<TaskControlBlockInner>,**  **}**  **pub struct TaskControlBlockInner {**  **//trap\_cx\_ppn 指出了应用地址空间中的 Trap 上下文被放在的物理页帧的物理页号。**  **pub trap\_cx\_ppn: PhysPageNum,**  **//base\_size 的含义是：应用数据仅有可能出现在应用地址空间低于 base\_size 字节的区域中。借助它我们可以清楚的知道应用有多少数据驻留在内存中**  **pub base\_size: usize,**  **//task\_cx 将暂停的任务的任务上下文保存在任务控制块中。**  **pub task\_cx: TaskContext,**  **//task\_status 维护当前进程的执行状态**  **pub task\_status: TaskStatus,**  **//memory\_set 表示应用地址空间。**  **pub memory\_set: MemorySet,**  **//parent 指向当前进程的父进程（如果存在的话）。注意我们使用 Weak 而非 Arc 来包裹另一个任务控制块，因此这个智能指针将不会影响父进程的引用计数。**  **pub parent: Option<Weak<TaskControlBlock>>,**  **//children 则将当前进程的所有子进程的任务控制块以 Arc 智能指针的形式保存在一个向量中，这样才能够更方便的找到它们。**  **pub children: Vec<Arc<TaskControlBlock>>,**  **//当进程调用 exit 系统调用主动退出或者执行出错由内核终止的时候，它的退出码 exit\_code 会被内核保存在它的任务控制块中，并等待它的父进程通过 waitpid 回收它的资源的同时也收集它的 PID 以及退出码。**  **pub exit\_code: i32,**  **}**  **//对于TaskControlBlockInner内部的字段的快捷访问**  **impl TaskControlBlockInner {**  **pub fn get\_trap\_cx(&self) -> &'static mut TrapContext {**  **self.trap\_cx\_ppn.get\_mut()**  **}**  **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 {**  **self.get\_status() == TaskStatus::Zombie**  **}**  **}**  **// os/src/task/task.rs**  **impl TaskControlBlock {**  **//inner\_exclusive\_access 通过 UPSafeCell<T>.exclusive\_access() 来得到一个 RefMut<'\_, TaskControlBlockInner> ，它可以被看成一个内层 TaskControlBlockInner 的可变引用并可以对它指向的内容进行修改。**  **pub fn inner\_exclusive\_access(&self) -> RefMut<'\_, TaskControlBlockInner> {**  **self.inner.exclusive\_access()**  **}**  **//getpid 以 usize 的形式返回当前进程的进程标识符。**  **pub fn getpid(&self) -> usize {**  **self.pid.0**  **}**  **//new 用来创建一个新的进程，目前仅用于内核中手动创建唯一一个初始进程 initproc 。**  **pub fn new(elf\_data: &[u8]) -> Self {...}**  **//exec 用来实现 exec 系统调用，即当前进程加载并执行另一个 ELF 格式可执行文件。**  **pub fn exec(&self, elf\_data: &[u8]) {...}**  **//fork 用来实现 fork 系统调用，即当前进程 fork 出来一个与之几乎相同的子进程。**  **pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {...}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/manager.rs |
| **主要功能:** | 任务管理器 |
| **源代码:**  **pub struct TaskManager {**  **ready\_queue: VecDeque<Arc<TaskControlBlock>>,**  **}**  **/// A simple FIFO scheduler.**  **// 将所有的任务控制块用引用计数 Arc 智能指针包裹后放在一个双端队列 VecDeque 中**  **impl TaskManager {**  **pub fn new() -> Self {**  **Self { ready\_queue: VecDeque::new(), }**  **}**  **//将一个任务加入队尾**  **pub fn add(&mut self, task: Arc<TaskControlBlock>) {**  **self.ready\_queue.push\_back(task);**  **}**  **//从队头中取出一个任务来执行**  **pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {**  **self.ready\_queue.pop\_front()**  **}**  **}**  **lazy\_static! {**  **pub static ref TASK\_MANAGER: UPSafeCell<TaskManager> = unsafe {**  **UPSafeCell::new(TaskManager::new())**  **};**  **}**  **pub fn add\_task(task: Arc<TaskControlBlock>) {**  **TASK\_MANAGER.exclusive\_access().add(task);**  **}**  **pub fn fetch\_task() -> Option<Arc<TaskControlBlock>> {**  **TASK\_MANAGER.exclusive\_access().fetch()**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/processor.rs |
| **主要功能:** | 处理器管理结构 |
| **源代码:**  **pub struct Processor {**  **//current 表示在当前处理器上正在执行的任务**  **current: Option<Arc<TaskControlBlock>>,**  **//idle\_task\_cx 表示当前处理器上的 idle 控制流的任务上下文。**  **idle\_task\_cx: TaskContext,**  **}**  **impl Processor {**  **pub fn new() -> Self {**  **Self {**  **current: None,**  **idle\_task\_cx: TaskContext::zero\_init(),**  **}**  **}**  **}**  **//创建单个 Processor 的全局实例 PROCESSOR**  **lazy\_static! {**  **pub static ref PROCESSOR: UPSafeCell<Processor> = unsafe {**  **UPSafeCell::new(Processor::new())**  **};**  **}**  **3 impl Processor {**  **//Processor::take\_current 可以取出当前正在执行的任务**  **4 pub fn take\_current(&mut self) -> Option<Arc<TaskControlBlock>> {**  **5 self.current.take()**  **6 }**  **// Processor::current 返回当前执行的任务的一份拷贝**  **7 pub fn current(&self) -> Option<Arc<TaskControlBlock>> {**  **8 self.current.as\_ref().map(|task| Arc::clone(task))**  **9 }**  **10 }**  **11//对 Processor::take\_current/current 进行封装并提供给内核其他子模块的接口**  **12 pub fn take\_current\_task() -> Option<Arc<TaskControlBlock>> {**  **13 PROCESSOR.exclusive\_access().take\_current()**  **14 }**  **15**  **16 pub fn current\_task() -> Option<Arc<TaskControlBlock>> {**  **17 PROCESSOR.exclusive\_access().current()**  **18 }**  **19 //基于 current\_task 实现，可以提供当前正在执行的任务的更多信息**  **20 pub fn current\_user\_token() -> usize {**  **21 let task = current\_task().unwrap();**  **22 let token = task.inner\_exclusive\_access().get\_user\_token();**  **23 token**  **24 }**  **25**  **26 pub fn current\_trap\_cx() -> &'static mut TrapContext {**  **27 current\_task().unwrap().inner\_exclusive\_access().get\_trap\_cx()**  **28 }**  **//idle 控制流，它运行在这个 CPU 核的启动栈上，功能是尝试从任务管理器中选出一个任务来在当前 CPU 核上执行。在内核初始化完毕之后，会通过调用 run\_tasks 函数来进入 idle 控制流：**  **3 pub fn run\_tasks() {**  **4 loop {**  **5 let mut processor = PROCESSOR.exclusive\_access();**  **6 if let Some(task) = fetch\_task() {**  **//得到 \_\_switch 的第一个参数，也就是当前 idle 控制流的 task\_cx\_ptr，这调用了Processor.get\_idle\_task\_cx\_ptr 方法。**  **7 let idle\_task\_cx\_ptr = processor.get\_idle\_task\_cx\_ptr();**  **8 // access coming task TCB exclusively**  **//要先获取从任务管理器中取出对应的任务控制块，并获取任务块内部的 next\_task\_cx\_ptr 作为 \_\_switch 的第二个参数，然后修改任务的状态为 Running**  **9 let mut task\_inner = task.inner\_exclusive\_access();**  **10 let next\_task\_cx\_ptr = &task\_inner.task\_cx as \*const TaskContext;**  **11 task\_inner.task\_status = TaskStatus::Running;**  **12 // stop exclusively accessing coming task TCB manually**  **//手动回收对即将执行任务的任务控制块的借用标记，使得后续我们仍可以访问该任务控制块**  **13 drop(task\_inner);**  **//修改当前 Processor 正在执行的任务为我们取出的任务**  **14 processor.current = Some(task);**  **15 // stop exclusively accessing processor manually**  **16 drop(processor);**  **17 unsafe {**  **//调用 \_\_switch 来从当前的 idle 控制流切换到接下来要执行的任务**  **18 \_\_switch(**  **19 idle\_task\_cx\_ptr,**  **20 next\_task\_cx\_ptr,**  **21 );**  **22 }**  **23 }**  **24 }**  **25 }**  **26**  **27 impl Processor {**  **28 fn get\_idle\_task\_cx\_ptr(&mut self) -> \*mut TaskContext {**  **29 &mut self.idle\_task\_cx as \*mut \_**  **30 }**  **31 }**  **//当一个应用用尽了内核本轮分配给它的时间片或者它主动调用 yield 系统调用交出 CPU 使用权之后，内核会调用 schedule 函数来切换到 idle 控制流并开启新一轮的任务调度**  **pub fn schedule(switched\_task\_cx\_ptr: \*mut TaskContext) {**  **let mut processor = PROCESSOR.exclusive\_access();**  **let idle\_task\_cx\_ptr = processor.get\_idle\_task\_cx\_ptr();**  **drop(processor);**  **unsafe {**  **\_\_switch(**  **switched\_task\_cx\_ptr,**  **idle\_task\_cx\_ptr,**  **);**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/task.rs |
| **主要功能:** | 实现TaskControlBlock |
| **源代码:**  **1 // os/src/task/task.rs**  **2**  **3 use super::{PidHandle, pid\_alloc, KernelStack};**  **4 use super::TaskContext;**  **5 use crate::config::TRAP\_CONTEXT;**  **6 use crate::trap::TrapContext;**  **7**  **8 // impl TaskControlBlock**  **9 pub fn new(elf\_data: &[u8]) -> Self {**  **10 // memory\_set with elf program headers/trampoline/trap context/user stack**  **11 let (memory\_set, user\_sp, entry\_point) = MemorySet::from\_elf(elf\_data);**  **12 let trap\_cx\_ppn = memory\_set**  **13 .translate(VirtAddr::from(TRAP\_CONTEXT).into())**  **14 .unwrap()**  **15 .ppn();**  **16 // alloc a pid and a kernel stack in kernel space**  **17 let pid\_handle = pid\_alloc();**  **18 let kernel\_stack = KernelStack::new(&pid\_handle);**  **19 let kernel\_stack\_top = kernel\_stack.get\_top();**  **20 // push a task context which goes to trap\_return to the top of kernel stack**  **21 let task\_control\_block = Self {**  **22 pid: pid\_handle,**  **23 kernel\_stack,**  **24 inner: unsafe { UPSafeCell::new(TaskControlBlockInner {**  **25 trap\_cx\_ppn,**  **26 base\_size: user\_sp,**  **27 task\_cx: TaskContext::goto\_trap\_return(kernel\_stack\_top),**  **28 task\_status: TaskStatus::Ready,**  **29 memory\_set,**  **30 parent: None,**  **31 children: Vec::new(),**  **32 exit\_code: 0,**  **33 })},**  **34 };**  **35 // prepare TrapContext in user space**  **36 let trap\_cx = task\_control\_block.inner\_exclusive\_access().get\_trap\_cx();**  **37 \*trap\_cx = TrapContext::app\_init\_context(**  **38 entry\_point,**  **39 user\_sp,**  **40 KERNEL\_SPACE.exclusive\_access().token(),**  **41 kernel\_stack\_top,**  **42 trap\_handler as usize,**  **43 );**  **44 task\_control\_block**  **45 }**  **3 impl TaskControlBlock {**  **4 pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {**  **5 // ---- access parent PCB exclusively**  **6 let mut parent\_inner = self.inner\_exclusive\_access();**  **7 // copy user space(include trap context)**  **//调用 MemorySet::from\_existed\_user 复制父进程地址空间得到子进程的地址空间**  **8 let memory\_set = MemorySet::from\_existed\_user(**  **9 &parent\_inner.memory\_set**  **10 );**  **11 let trap\_cx\_ppn = memory\_set**  **12 .translate(VirtAddr::from(TRAP\_CONTEXT).into())**  **13 .unwrap()**  **14 .ppn();**  **15 // alloc a pid and a kernel stack in kernel space**  **16 let pid\_handle = pid\_alloc();**  **17 let kernel\_stack = KernelStack::new(&pid\_handle);**  **18 let kernel\_stack\_top = kernel\_stack.get\_top();**  **19 let task\_control\_block = Arc::new(TaskControlBlock {**  **20 pid: pid\_handle,**  **21 kernel\_stack,**  **22 inner: unsafe { UPSafeCell::new(TaskControlBlockInner {**  **23 trap\_cx\_ppn,**  **//让子进程和父进程的 base\_size ，即应用数据的大小保持一致**  **24 base\_size: parent\_inner.base\_size,**  **25 task\_cx: TaskContext::goto\_trap\_return(kernel\_stack\_top),**  **26 task\_status: TaskStatus::Ready,**  **27 memory\_set,**  **//将父进程的弱引用计数放到子进程的进程控制块中**  **28 parent: Some(Arc::downgrade(self)),**  **29 children: Vec::new(),**  **30 exit\_code: 0,**  **31 })},**  **32 });**  **//将子进程插入到父进程的孩子向量 children 中**  **33 // add child**  **34 parent\_inner.children.push(task\_control\_block.clone());**  **35 // modify kernel\_sp in trap\_cx**  **36 // \*\*\*\* access children PCB exclusively**  **37 let trap\_cx = task\_control\_block.inner\_exclusive\_access().get\_trap\_cx();**  **38 trap\_cx.kernel\_sp = kernel\_stack\_top;**  **39 // return**  **40 task\_control\_block**  **41 // ---- stop exclusively accessing parent/children PCB automatically**  **42 }**  **43 }** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/mod.rs |
| **主要功能:** | 实现任务管理 |
| **源代码:**  **use processor::{task\_current\_task, schedule};**  **use manager::add\_task;**  **pub fn suspend\_current\_and\_run\_next() {**  **// There must be an application running.**  **//通过 take\_current\_task 来取出当前正在执行的任务，修改其进程控制块内的状态，随后将这个任务放入任务管理器的队尾**  **let task = take\_current\_task().unwrap();**  **// ---- access current TCB exclusively**  **let mut task\_inner = task.inner\_exclusive\_access();**  **let task\_cx\_ptr = &mut task\_inner.task\_cx as \*mut TaskContext;**  **// Change status to Ready**  **task\_inner.task\_status = TaskStatus::Ready;**  **drop(task\_inner);**  **// ---- stop exclusively accessing current PCB**  **// push back to ready queue.**  **add\_task(task);**  **// jump to scheduling cycle**  **//调用 schedule 函数来触发调度并切换任务**  **schedule(task\_cx\_ptr);**  **}**  **11 pub fn exit\_current\_and\_run\_next(exit\_code: i32) {**  **12 //调用 take\_current\_task 来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，这是为了正确维护进程控制块的引用计数**  **13 let task = take\_current\_task().unwrap();**  **14 // \*\*\*\* access current TCB exclusively**  **15 let mut inner = task.inner\_exclusive\_access();**  **16 // Change status to Zombie**  **//将进程控制块中的状态修改为 TaskStatus::Zombie 即僵尸进程，这样它后续才能被父进程在 waitpid 系统调用的时候回收**  **17 inner.task\_status = TaskStatus::Zombie;**  **18 // 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**  **19 inner.exit\_code = exit\_code;**  **20 // do not move to its parent but under initproc**  **21**  **22 // ++++++ access initproc TCB exclusively**  **//将当前进程的所有子进程挂在初始进程 initproc 下面，其做法是遍历每个子进程，修改其父进程为初始进程，并加入初始进程的孩子向量中。第 32 行将当前进程的孩子向量清空**  **23 {**  **24 let mut initproc\_inner = INITPROC.inner\_exclusive\_access();**  **25 for child in inner.children.iter() {**  **26 child.inner\_exclusive\_access().parent = Some(Arc::downgrade(&INITPROC));**  **27 initproc\_inner.children.push(child.clone());**  **28 }**  **29 }**  **30 // ++++++ stop exclusively accessing parent PCB**  **31**  **32 inner.children.clear();**  **33 // deallocate user space**  **//对于当前进程占用的资源进行早期回收**  **34 inner.memory\_set.recycle\_data\_pages();**  **35 drop(inner);**  **36 // \*\*\*\* stop exclusively accessing current PCB**  **37 // drop task manually to maintain rc correctly**  **38 drop(task);**  **39 // we do not have to save task context**  **40 let mut \_unused = TaskContext::zero\_init();**  **//调用 schedule 触发调度及任务切换**  **41 schedule(&mut \_unused as \*mut \_);**  **42 }** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/mm/memory\_set.rs |
| **主要功能:** | 实现 [`MapArea`] 和[`MemorySet`] |
| **源代码:**  **1 // os/src/mm/memory\_set.rs**  **2**  **3 impl MapArea {**  **//MapArea::from\_another 可以从一个逻辑段复制得到一个虚拟地址区间、映射方式和权限控制均相同的逻辑段，不同的是由于它还没有真正被映射到物理页帧上，所以 data\_frames 字段为空。**  **4 pub fn from\_another(another: &MapArea) -> Self {**  **5 Self {**  **6 vpn\_range: VPNRange::new(**  **7 another.vpn\_range.get\_start(),**  **8 another.vpn\_range.get\_end()**  **9 ),**  **10 data\_frames: BTreeMap::new(),**  **11 map\_type: another.map\_type,**  **12 map\_perm: another.map\_perm,**  **13 }**  **14 }**  **15 }**  **16**  **17 impl MemorySet {**  **//MemorySet::from\_existed\_user 可以复制一个完全相同的地址空间**  **18 pub fn from\_existed\_user(user\_space: &MemorySet) -> MemorySet {**  **//通过 new\_bare 新创建一个空的地址空间**  **19 let mut memory\_set = Self::new\_bare();**  **20 // map trampoline**  **//通过 map\_trampoline 为这个地址空间映射上跳板页面**  **21 memory\_set.map\_trampoline();**  **22 // copy data sections/trap\_context/user\_stack**  **23 for area in user\_space.areas.iter() {**  **24 let new\_area = MapArea::from\_another(area);**  **25 memory\_set.push(new\_area, None);**  **26 // copy data from another space**  **27 for vpn in area.vpn\_range {**  **28 let src\_ppn = user\_space.translate(vpn).unwrap().ppn();**  **29 let dst\_ppn = memory\_set.translate(vpn).unwrap().ppn();**  **30 dst\_ppn.get\_bytes\_array().copy\_from\_slice(src\_ppn.get\_bytes\_array());**  **31 }**  **32 }**  **33 memory\_set**  **34 }**  **35 }** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/mm/page\_table.rs |
| **主要功能:** | Implementation of [`PageTableEntry`] and [`PageTable`]. |
| **源代码:**  **3 pub fn translated\_str(token: usize, ptr: \*const u8) -> String {**  **4 let page\_table = PageTable::from\_token(token);**  **5 let mut string = String::new();**  **6 let mut va = ptr as usize;**  **7 loop {**  **8 let ch: u8 = \*(page\_table.translate\_va(VirtAddr::from(va)).unwrap().get\_mut());**  **9 if ch == 0 {**  **10 break;**  **11 } else {**  **12 string.push(ch as char);**  **13 va += 1;**  **14 }**  **15 }**  **16 string**  **17 }**  **18 //调用 translated\_str 找到要执行的应用名并试图在应用加载器提供的 get\_app\_data\_by\_name 接口中找到对应的 ELF 格式的数据**  **19 // os/src/syscall/process.rs**  **20**  **21 pub fn sys\_exec(path: \*const u8) -> isize {**  **22 let token = current\_user\_token();**  **23 let path = translated\_str(token, path);**  **24 if let Some(data) = get\_app\_data\_by\_name(path.as\_str()) {**  **25 let task = current\_task().unwrap();**  **26 task.exec(data);**  **27 0**  **28 } else {**  **29 -1**  **30 }**  **31 }** | |

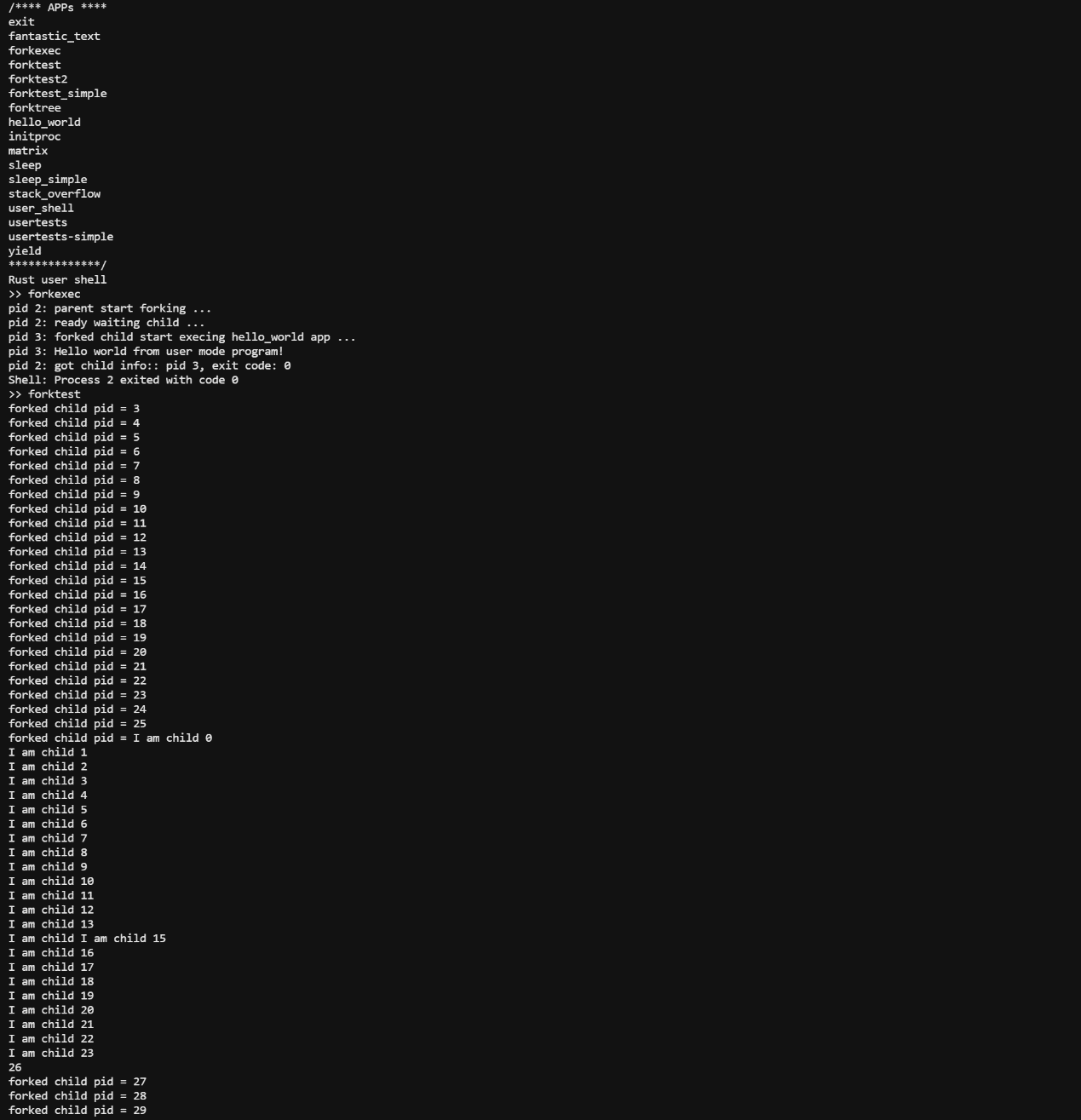
|  |  |
| --- | --- |
| **文件名:** | os/src/trap/mod.rs |
| **主要功能:** | trap处理功能 |
| **源代码:**  **#[no\_mangle]**  **//对于系统调用 sys\_exec 来说，一旦调用它之后，我们会发现 trap\_handler 原来上下文中的 cx 失效了——因为它是用来访问之前地址空间中 Trap 上下文被保存在的那个物理页帧的，而现在它已经被回收掉了。因此，为了能够处理类似的这种情况，我们在 syscall 分发函数返回之后需要重新获取 cx**  **pub fn trap\_handler() -> ! {**  **set\_kernel\_trap\_entry();**  **let scause = scause::read();**  **let stval = stval::read();**  **match scause.cause() {**  **Trap::Exception(Exception::UserEnvCall) => {**  **// jump to next instruction anyway**  **let mut cx = current\_trap\_cx();**  **cx.sepc += 4;**  **// get system call return value**  **let result = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]);**  **// cx is changed during sys\_exec, so we have to call it again**  **cx = current\_trap\_cx();**  **cx.x[10] = result as usize;**  **}**  **...**  **}**  **trap\_return();**  **}** | |

# 4 运行结果与分析

待内核初始化完毕之后，将在屏幕上打印可用的应用列表并进入 Shell 程序

其中 usertests 打包了很多应用，只要执行它就能够自动执行一系列应用

只需输入应用的名称并回车即可在系统中执行该应用。以应用 exit 为例：





应用执行完毕后，将继续回到 Shell 程序的命令输入模式

# 5 实验总结

我们开发了一个命令行应用，形成了用户与操作系统进行交互的命令行界面。它在启动后，会加载执行用户态的 Shell 程序，并可以通过 Shell 程序提供的命令行交互界面，让使用者敲入要执行的应用程序名字，就可以创建一个子进程来执行这个应用程序，实现了灵活的人机交互和进程管理的动态灵活性。

本章完成的操作系统的核心目标就是： 让开发者能够控制程序的运行 。