第十讲：进程和线程控制

第 5 节：rCore 进程和线程控制

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

1. 第 5 节：rCore 进程和线程控制
   - 进程和线程控制块
   - 线程状态转换
   - 线程上下文切换
   - 进程和线程控制接口

rCore 进程控制块结构

rCore/kernel/src/process/structs.rs

```rust
pub struct Process {
    // resources
    pub vm: Arc<Mutex<MemorySet>>,
    pub files: BTreeMap<usize, FileLike>,
    pub cwd: String,
    pub exec_path: String,
    futexes: BTreeMap<usize, Arc<Condvar>>,
    pub semaphores: SemProc,

    // relationship
    pub pid: Pid, // i.e. tgid, usually the tid of first thread
    pub parent: Weak<Mutex<Process>>,
    pub children: Vec<Weak<Mutex<Process>>>,
    pub threads: Vec<Tid>, // threads in the same process

    // for waiting child
    pub child_exit: Arc<Condvar>, // notified when the a child process is going to terminate
    pub child_exit_code: BTreeMap<usize, usize>, // child process store its exit code here
}
```

rCore 内存地址空间结构

rCore/kernel/src/memory.rs

```
pub type MemorySet = rcore_memory::memory_set::MemorySet<PageTableImpl>;
```

rCore/kernel/src/arch/riscv/paging.rs

```
pub struct PageTableImpl {
    page_table: TopLevelPageTable<'static>,
    root_frame: Frame,
    entry: Option<PageEntry>,
}
```

rCore 内存地址空间结构

rCore/crate/memory/src/memory_set/mod.rs

```
type TopLevelPageTable<'a> = riscv::paging::Rv32PageTable<'a>;
```

riscv/src/paging/multi_level.rs

```
pub struct Rv32PageTable<'a> {
    root_table: &'a mut PageTable,
    linear_offset: usize, // VA = PA + linear_offset
}
```

rCore 内存地址空间结构

rCore/crate/memory/src/paging/mod.rs

```rust
/// Activate this page table
unsafe fn activate(&self) {
    let old_token = Self::active_token();
    let new_token = self.token();
    debug!("switch table {:x?} -> {:x?}", old_token, new_token);
    if old_token != new_token {
        Self::set_token(new_token);
        Self::flush_tlb();
    }
}
```

rCore/kernel/src/arch/riscv/paging.rs

```rust
unsafe fn set_token(token: usize) {
    asm!("csrw satp, $0" :: "r"(token) :: "volatile");
```

rCore 线程控制块

rCore/kernel/src/process/structs.rs

```rust
struct Thread {
    /// Current status of the thread.
    status: Status,
    /// Next status after the thread stop running.
    status_after_stop: Status,
    /// A waiter thread of this. It will be woken up on my exit.
    waiter: Option<Tid>,
    /// If detached, all resources will be released on exit.
    detached: bool,
    /// The context of the thread.
    context: Option<Box<dyn Context>>,
}
```

线程状态数据结构

rcore-thread/src/thread_pool.rs

```
pub enum Status {
    Ready,
    Running(usize),
    Sleeping,
    /// aka ZOMBIE. Its context was dropped.
    Exited(ExitCode),
}
```

线程状态转换

rcore-thread/src/thread_pool.rs

```
fn set_status(&self, tid: Tid, status: Status)
```

```
match (&proc.status, &status) {
    (Status::Ready, Status::Ready) => return,
    (Status::Ready, _) => self.scheduler.remove(tid),
    (Status::Exited(_), _) => panic!("can not set status for a exited thread"),
    (Status::Sleeping, Status::Exited(_)) => self.timer.lock().stop(Event::Wakeup(tid)),
    (Status::Running(_), Status::Ready) => {} // thread will be added to scheduler in stop()
    (_, Status::Ready) => self.scheduler.push(tid),
    _ => {}
}
match proc.status {
    Status::Running(_) => proc.status_after_stop = status,
    _ => proc.status = status,
}
match proc.status {
    Status::Exited(_) => self.exit_handler(proc_lock),
    _ => {}
}
```

线程上下文切换数据结构

```rust
pub struct TrapFrame {
    /// General registers
    pub x: [usize; 32],
    /// Supervisor Status
    pub sstatus: Sstatus,
    /// Supervisor Exception Program Counter
    pub sepc: usize,
    /// Supervisor Trap Value
    pub stval: usize,
    /// Supervisor Cause
    pub scause: Scause,
}
```

在陷入异常时向栈中压入的内容，由 trap.S 的 __alltraps 构建。

线程上下文切换数据结构

```
struct ContextData {
    /// Return address
    ra: usize,
    /// Page table token
    satp: usize,
    /// Callee-saved registers
    s: [usize; 12],
}
```

执行上下文切换时向栈中压入的内容，由 *__switch*() 函数构建。

线程上下文切换数据结构

```
pub struct InitStack {
    context: ContextData,
    tf: TrapFrame,
}
```

对于新创建的线程，不仅要向栈中压入 ContextData 结构，还需手动构造 TrapFrame 结构。为了方便管理就定义了 InitStack 包含这两个结构体。

线程上下文切换数据结构

```rust
pub struct Context {
    /// The stack pointer of the suspended thread.
    /// A `ContextData` is stored here.
    sp: usize,
}
```

每个进程控制块 Process 都会维护一个平台相关的 Context 对象。

切换函数

rCore/kernel/src/process/structs.rs

```
unsafe fn switch_to(&mut self, target: &mut dyn rcore_thread::Context) {
    use core::mem::transmute;
    let (target, _): (&mut Thread, *const ()) = transmute(target);
    self.context.switch(&mut target.context);
}
```

切换函数

rCore/kernel/src/arch/riscv/context.rs

```
/// Switch to another kernel thread.
///
/// Push all callee-saved registers at the current kernel stack.
/// Store current sp, switch to target.
/// Pop all callee-saved registers, then return to the target.
#[naked]
#[inline(never)]
pub unsafe extern "C" fn switch(&mut self, _target: &mut Self) {
    #[cfg(target_arch = "riscv32")]
    asm!(
        r"
    .equ XLENB, 4
    .macro Load reg, mem
        lw \reg, \mem
    .endm
    .macro Store reg, mem
        sw \reg, \mem
    .endm"
    );
```

线程切换过程

rcore-thread/src/processor.rs

```rust
/// Called by timer interrupt handler.
///
/// The interrupt should be disabled in the handler.
pub fn tick(&self) {
    // If I'm idle, tid == None, need_reschedule == false.
    // Will go back to `run()` after interrupt return.
    let tid = self.inner().thread.as_ref().map(|p| p.0);
    let need_reschedule = self.manager().tick(self.inner().id, tid);
    if need_reschedule {
        self.yield_now();
    }
}
```

线程切换过程

rcore-thread/src/scheduler/mod.rs

```rust
/// The scheduler for a ThreadPool
pub trait Scheduler: 'static {
    /// Push a thread to the back of ready queue.
    fn push(&self, tid: Tid);
    /// Select a thread to run, pop it from the queue.
    fn pop(&self, cpu_id: usize) -> Option<Tid>;
    /// Got a tick from CPU.
    /// Return true if need reschedule.
    fn tick(&self, current_tid: Tid) -> bool;
    /// Set priority of a thread.
    fn set_priority(&self, tid: Tid, priority: u8);
    /// remove a thread in ready queue.
    fn remove(&self, tid: Tid);
}
```

进程管理的系统调用

rCore/kernel/src/syscall/proc.rs

proc.rs rCore • kernel/src/syscall 15

```
pub fn sys_fork(&mut self) -> SysResult {
pub fn sys_vfork(&mut self) -> SysResult {
pub fn sys_clone(
pub fn sys_wait4(&mut self, pid: isize, wstatus: *mut i32) -> SysResult {
pub fn sys_exec(
pub fn sys_yield(&mut self) -> SysResult {
pub fn sys_kill(&mut self, pid: usize, sig: usize) -> SysResult {
pub fn sys_getpid(&mut self) -> SysResult {
pub fn sys_gettid(&mut self) -> SysResult {
pub fn sys_getppid(&mut self) -> SysResult {
pub fn sys_exit(&mut self, exit_code: usize) -> ! {
pub fn sys_exit_group(&mut self, exit_code: usize) -> ! {
pub fn sys_nanosleep(&mut self, req: *const TimeSpec) -> SysResult {
pub fn sys_set_priority(&mut self, priority: usize) -> SysResult {
pub fn sys_set_tid_address(&mut self, tidptr: *mut u32) -> SysResult {
```

内核中线程模块接口

rcore-thread/src/std_thread.rs

std_thread.rs rcore-thread • src 10

```
pub fn current() -> Thread {
pub fn sleep(dur: Duration) {
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
pub fn yield_now() {
pub fn park() {
pub fn park_action(f: impl FnOnce()) {
pub fn unpark(&self) {
pub fn id(&self) -> usize {
pub fn thread(&self) -> &Thread {
pub fn join(self) -> Result<T, ()> {
```