武漢大學
WUHAN UNIVERSITY

第七节
进程管理与调度

Implementing printf and Screen Clearing in the Kernel

《操作系统实践A》

07 实验目标

武漢大學
WUHAN UNIVERSITY

- 深入理解 操作系统如何创建、管理和调度进程
- 分析 xv6 源码中的进程生命周期管理机制
- 掌握 调度器 (scheduler) 的工作原理和简单调度算法实现
- 完善对 进程状态转换、内存分配与释放 的理解
- 学会阅读、修改并调试 xv6 内核代码

07 核心学习资料

- 《操作系统概念》第 3–5 章：进程、线程、CPU 调度
- xv6 手册 第 2–4 章：操作系统组织、页表管理、陷阱与系统调用
- RISC- V 调用约定
  - https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

07 xv6进程管理源码分析

- kernel/proc.h - 进程结构体定义
  - 重点： struct proc 的字段含义和生命周期
- kernel/proc.c - 进程管理核心函数
  - 重点函数： allocproc() , fork() , exit() , wait() , scheduler()
  - 学习要点：进程状态转换、内存管理、调度策略
- kernel/swtch.S - 上下文切换汇编代码
  - 理解：寄存器保存策略、栈切换机制
- kernel/sysproc.c - 进程相关系统调用
  - 重点： sys_fork() , sys_exit() , sys_wait() , sys_kill()

07 任务1：深入理解进程抽象

- 1. 分析xv6的进程结构体：

```
struct proc {
        struct spinlock lock;
        enum procstate state; // 进程状态
        void *chan; // 等待通道
        int killed; // 是否被杀死
        int xstate; // 退出状态
        int pid; // 进程ID
        pagetable_t pagetable; // 用户页表
        struct trapframe *trapframe; // 陷阱帧
        struct context context; // 调度上下文
        // ...更多字段
};
```

- 每个字段的作用是什么?
- 进程状态转换图是怎样的?
- 为什么需要锁保护?

07 任务1：深入理解进程抽象

- 2. 理解进程生命周期：
  - UNUSED → USED → RUNNABLE → RUNNING → SLEEPING → ZOMBIE

状态名	含义	典型场景
UNUSED	空闲，可被分配	刚启动时的空槽
USED	已分配但未运行	刚创建但未就绪
RUNNABLE	可被调度运行	fork 后的子进程
RUNNING	正在 CPU 上执行	内核调度选中
SLEEPING	等待事件或资源	调用 sleep(chan)
ZOMBIE	已退出，待父进程回收	exit() 之后

  - 每个状态转换的触发条件是什么？
  - 哪些操作需要原子性保护？

07 任务1：深入理解进程抽象

- 深入思考：
  - 为什么需要ZOMBIE状态?
  - 进程表的大小限制有什么影响?
  - 如何防止进程ID重复?

07 任务2：分析xv6的进程创建机制

- 1. 研读 allocproc() 函数：
  - 如何在进程表中找到空闲槽位?
  - 进程ID是如何分配的?
  - 用户栈是如何设置的?
  - 陷阱帧的初始化过程

操作	说明
查找空槽	遍历 ptable，找到 state == UNUSED 的进程项
分配 PID	使用全局变量 nextpid++，一般为递增计数
设置栈空间	proc->kstack = kalloc() 为内核栈预留空间
初始化陷阱帧	将 trapframe 指针指向内核栈顶，用于保存用户寄存器状态
返回 proc 指针	供 fork() 使用

07 任务2：分析xv6的进程创建机制

- 2. 深入理解 fork() 实现：
- 父子进程返回值不同的原因
  - 父进程：收到子进程 PID
  - 子进程：返回 0
  - 便于两者在相同代码执行处做不同逻辑判断
- 内存复制原理
  - xv6 使用 uvmcopy() 将父进程页表复制给子进程
  - 复制包括：代码段、数据段、堆栈区
- 失败清理机制
  - 若任一初始化失败（如内存不足），调用 freeproc() 回收已分配资源
  - 保证系统资源不泄漏

```
int fork(void) {
    // 1. 分配新进程结构
    // 2. 复制用户内存
    // 3. 复制陷阱帧
    // 4. 设置返回值
    // 5. 标记为RUNNABLE
}
```

07 任务2：分析xv6的进程创建机制

- 3. 分析进程退出机制：
  - exit() 与 wait() 的协作关系

阶段	说明
exit()	当前进程停止运行，释放打开的文件与内核端资源
标记 ZOMBIE	保存退出码，等待父进程调用 wait()
wait()	父进程收集子进程退出状态，并释放其 proc 结构

  - 资源回收的时机和方式
  - 关闭文件描述符：
    - fileclose(f)->释放页表、内核栈内存（增强稳定性：避免“僵尸进程泄露）
  - 孤儿进程的处理：
    - 当父进程在子进程退出前终止→ 子进程的父指针指向 initproc
    - initproc 会周期性调用 wait() → 自动回收所有孤儿进程（防止僵尸积累）

07 任务2：分析xv6的进程创建机制

- 关键问题：
- fork()的性能瓶颈在哪里？
  - 当父进程内存空间较大时，复制所有内存页面是否必要？
  - 若子进程立即调用 `exec()`，复制的内存会怎样？
  - CPU 与内存带宽是否会成为瓶颈？
  - 提示：xv6 的 `fork()` 采用逐页 `memmove()`，时间复杂度约为 `O(n)`（n = 页数）
- 如何实现写时复制优化？
  - 页表项都先标记为只读 (read-only)
  - 当发生写操作时触发页错误 (page fault)
  - 内核在异常处理程序中复制目标页
  - 修改页表，恢复写权限

07 任务3：设计你的进程管理系统

- 设计要求：
  - 1. 确定进程结构体设计
  - 2. 选择合适的进程表组织方式
  - 3. 设计进程ID分配策略
- 核心接口设计：

```
// 进程管理基本接口
struct proc* alloc_process(void); // 分配进程结构
void free_process(struct proc *p); // 释放进程资源
int create_process(void (*entry)(void)); // 创建新进程
void exit_process(int status); // 终止当前进程
int wait_process(int *status); // 等待子进程
```

你需要考虑的设计问题：

- 1. 进程表用数组还是链表?
- 2. 如何高效查找特定PID的进程?
- 3. 是否需要进程组和会话的概念?
- 4. 如何处理进程资源限制?

实现策略：

- 1. 先实现基本的进程创建和销毁
- 2. 再添加父子关系管理
- 3. 最后考虑性能优化

07 任务4：实现上下文切换机制

- 参考xv6的swtch.S，理解：
- 1. 上下文切换的本质：
  - 在 xv6 中，`swtch.S` 负责保存当前进程的 CPU 状态并切换到另一进程。
  - 上下文切换 = 保存 + 恢复 内核寄存器现场
  - 当前进程 context： 寄存器 → 保存到内核栈
  - 切换后进程 context： 寄存器 ← 从内核栈恢复
- 哪些寄存器需要保存?

寄存器	说明
ra (return address)	返回地址寄存器
sp (stack pointer)	栈指针
s0–s11	被调用者保存的寄存器（callee-saved）

07 任务4：实现上下文切换机制

- 为什么不保存所有寄存器?
  - 临时寄存器（t0–t6）由 调用者 负责保存；
  - 上下文切换时，只需保存能够保持过程状态的寄存器；
  - 减少切换开销，提升效率。
- 调用者保存 vs 被调用者保存的区别

分类	负责保存的对象	常用寄存器	使用场景
调用者保存 (caller-saved)	当前函数	t0–t6, a0–a7	临时变量、函数参数
被调用者保存 (callee-saved)	被调用函数	s0–s11, sp, ra	函数内部上下文

07 任务4：实现上下文切换机制

- 实现挑战：

```
// 上下文结构体设计
    struct context {
    uint64 ra; // 返回地址
    uint64 sp; // 栈指针
    // 需要保存哪些其他寄存器?
    // 为什么这样选择?
};

// 上下文切换函数
void swtch(struct context *old, struct context *new);
```

- 关键技术点：
  - 上下文切换必须是原子操作
  - 中断状态的管理
  - 多级栈的处理

07 任务5：实现调度器

- 参考xv6的调度策略：
- 1. 分析 scheduler() 函数
- **轮转调度的实现方式**：简单循环遍历进程表，按顺序挑选 RUNNABLE 进程，每个进程分得一个时间片（time slice），时间片到期 → 时钟中断触发 抢占切换，公平性优先，适合教学验证
- **如何避免忙等待？** 进程在 sleep() 时会释放锁并挂起自己。wakeup() 唤醒后再回到就绪队列。CPU 若无可运行进程，可进入低功耗等待。避免忙等待的关键： 利用 sleep() + wakeup() 实现阻塞式等待机制。
- **为什么需要开启中断？** 调度器主动调用 sti() 开启中断响应。保证 CPU 能响应 时钟中断、I/O 中断 等事件。时钟中断用于周期性触发调度 → 实现 抢占式多任务。没有中断，就无法在进程内部自动让出 CPU！

07 任务5：实现调度器

- 理解调度时机：
  - 主动调度 vs 抢占调度

调度类型	触发方式	典型函数
主动调度	进程自愿让出 CPU	yield()
抢占调度	时钟中断触发	trap() → yield()

  - yield() 函数的作用：将自己状态改为 RUNNABLE，触发 swtch() 回到 scheduler()
  - 时钟中断如何触发调度

    每次时钟中断（通常 10–100Hz）

    → 进入 trap()

    → 检查当前进程是否用尽时间片

    → 调用 yield() 实现抢占

07 任务5：实现调度器

- 调度器设计考虑：
  - 1. 如何选择下一个运行的进程?
  - 2. 如何处理优先级?
  - 3. 如何避免饥饿?
  - 4. 如何平衡公平性和效率?
- 扩展调度算法：
  - 优先级调度
  - 多级反馈队列
  - 完全公平调度器(CFS)

```
void scheduler(void) {
        struct proc *p;
        struct cpu *c = mycpu();
        c->proc = 0;
        for(;;) {
                // 开启中断，允许设备中断
                intr_on();
                for(p = proc; p < &proc[NPROC]; p++) {
                        acquire(&p->lock);
                        if(p->state == RUNNABLE) {
                                // 找到可运行进程，切换过去
                                p->state = RUNNING;
                                c->proc = p;
                                swtch(&c->context, &p->context);
                                c->proc = 0;
                        }
                        release(&p->lock);
                }
        }
}
```

07 任务6：实现进程同步原语

- 基于xv6的sleep/wakeup机制：
- 1. 理解条件变量的概念：

```
// 等待条件满足
void sleep(void *chan, struct spinlock *lk);

// 唤醒等待特定条件的进程
void wakeup(void *chan);
```

- 2. 分析典型使用模式：
  - a. 生产者-消费者问题
  - b. 读者-写者问题
  - c. 信号量的实现
- 实现要点：
  - 避免lost wakeup问题
  - 锁的正确使用
  - 中断状态的管理

要点	说明
避免 lost wakeup	唤醒信号需在持锁状态下操作，防止错过唤醒
锁的使用	sleep() 会先释放传入的锁，再进入睡眠，唤醒后重新加锁
中断状态管理	调用时保持中断关闭，防止竞争条件和中断干扰

- 进程创建测试

```
void test_process_creation(void) {
    printf("Testing process creation...\n");

    // 测试基本的进程创建
    int pid = create_process(simple_task);
    assert(pid > 0);

    // 测试进程表限制
    int pids[NPROC];
    int count = 0;
    for (int i = 0; i < NPROC + 5; i++) {
        int pid = create_process(simple_task);
        if (pid > 0) {
            pids[count++] = pid;
        } else {
            break;
        }
    }
    printf("Created %d processes\n", count);

    // 清理测试进程
    for (int i = 0; i < count; i++) {
        wait_process(NULL);
    }
}
```

- 调度器测试

```
void test_scheduler(void) {
    printf("Testing scheduler...\n");

    // 创建多个计算密集型进程
    for (int i = 0; i < 3; i++) {
        create_process(cpu_intensive_task);
    }

    // 观察调度行为
    uint64 start_time = get_time();
    sleep(1000); // 等待1秒
    uint64 end_time = get_time();

    printf("Scheduler test completed in %lu cycles\n",
    end_time - start_time);
}
```

- 同步机制测试

```
void test_synchronization(void) {
    // 测试生产者-消费者场景
    shared_buffer_init();

    create_process(producer_task);
    create_process(consumer_task);

    // 等待完成
    wait_process(NULL);
    wait_process(NULL);

    printf("Synchronization test completed\n");
}
```

- 进程状态调试

```
void debug_proc_table(void) {
    printf("=== Process Table ===\n");
    for (int i = 0; i < NPROC; i++) {
        struct proc *p = &proc[i];
        if (p->state != UNUSED) {
            printf("PID:%d State:%d Name:%s\n",
            p->pid, p->state, p->name);
        }
    }
}
```

07 测试与调试策略

- 调度器调试
  - 在调度器中添加统计信息
  - 跟踪进程运行时间
  - 分析调度延迟
- 内存泄漏检测
  - 跟踪进程创建和销毁
  - 检查页表释放
  - 监控进程表使用情况

07 思考题

- 1. 进程模型：
  - 为什么选择这种进程结构设计？
  - 如何支持轻量级线程？
- 2. 调度策略：
  - 轮转调度的公平性如何？
  - 如何实现实时调度？
- 3. 性能优化：
  - fork()的性能瓶颈如何解决？
  - 上下文切换开销如何降低？
- 4. 资源管理：
  - 如何实现进程资源限制？
  - 如何处理进程资源泄漏？
- 5. 扩展性：
  - 如何支持多核调度？
  - 如何实现负载均衡？

武漢大學
WUHAN UNIVERSITY

End of This Part