武汉大学计算机学院
本科生课程设计报告

实验五：进程管理与调度

专 业 名 称　：计算机科学与技术

课 程 名 称　：操作系统实践 A

指 导 教 师　：李祖超　副教授

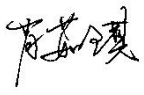
学 生 学 号　：2023302111416

学 生 姓 名　：肖茹琪

二〇二五年十一月

郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：  　　日期：2025.11.2

摘　要

本次实验旨在深入理解操作系统的核心抽象——进程，并基于 RISC-V 架构设计实现了支持多任务并发的进程管理子系统。实验首先从数据结构入手，定义了进程控制块（PCB/struct proc），用以此描述进程的状态（UNUSED, RUNNABLE, RUNNING 等）、内核栈及上下文信息。其次，重点攻克了进程上下文切换这一技术难点，通过编写汇编代码 swtch.S，实现了寄存器级的现场保存与恢复，完成了内核线程间的控制流切换。在此基础上，实现了完整的进程生命周期管理，包括进程创建、内存与状态复制、资源回收以及父进程等待。

针对核心的 CPU 调度机制，实验构建了基于时间片轮转策略的调度器 scheduler。该调度器能够遍历进程表，选取就绪状态的进程并分配 CPU 资源，配合时钟中断实现了抢占式多任务处理。最后，通过进程并发创建与调度测试，验证了系统的稳定性与调度算法的正确性。本次实验成功将操作系统从单一执行流扩展为多进程并发环境，标志着内核具备了管理复杂计算任务的能力。

关键词：PCB；上下文切换；进程调度；生命周期管理；时间片轮转

目录

1 实验目的和意义

1.1 实验目的

本实验旨在通过从零构建进程管理系统，深入剖析现代操作系统实现多任务并发的核心机制。通过分析 xv6-riscv 源码并结合理论知识，具体达成以下目标：

1. 掌握进程抽象与状态机模型： 深入理解 struct proc 结构体各字段的物理含义，掌握进程在 UNUSED, RUNNABLE, RUNNING, ZOMBIE 等状态之间转换的触发条件与逻辑链条。

2. 理解并实现上下文切换： 掌握 RISC-V 架构下的函数调用约定，通过编写 swtch.S 汇编代码，精确控制程序计数器（ra）和栈指针（sp）的切换，理解“线程”本质上是寄存器状态集合的原理。

3. 构建完整的进程生命周期管理： 独立实现进程的创建（fork 机制中的资源拷贝）、执行、退出（exit 中的资源释放）及回收（wait 机制）的全流程。

4. 实现 CPU 调度算法： 设计并实现一个基于状态轮询的调度器框架，理解 scheduler 函数如何与 yield 及 sched 函数配合，在时钟中断驱动下实现 CPU 时间片的公平分配。

5. 处理进程间的同步与竞争： 在操作共享的进程表时，学会使用自旋锁保护临界区，防止多核环境下可能出现的数据竞争与状态不一致问题。

1.2 实验意义

进程管理是操作系统将硬件资源虚拟化并提供给应用程序使用的关键步骤，本实验对于理解计算机系统如何“同时”做多件事具有里程碑式的意义。

首先，通过实现上下文切换，打破了程序必须从头执行到尾的线性思维，建立了对“并发执行”和“虚拟化 CPU”的底层认知。这种在毫秒级时间内快速切换执行流的技术，是现代分时操作系统的基石。其次，进程生命周期管理的实现，让内核从单纯的事件响应者变成了资源的管理者，特别是 fork 和 exec 机制的设计，展示了操作系统如何高效地复用代码与数据。此外，调度器的设计直接关

系到系统的吞吐量与响应速度，通过编写调度算法，能够直观体会算法策略对系统性能的决定性影响。最后，本实验中涉及的大量指针操作、栈空间管理以及汇编与 C 语言的混合编程，极大地锻炼了对计算机底层内存布局和执行模型的掌控能力，为后续实现更复杂的文件系统和虚拟内存管理奠定了坚实基础。

2 实验准备

在着手编写进程管理系统的核心代码之前，必须透彻理解操作系统如何抽象“进程”这一概念，以及 xv6 内核是如何通过精巧的数据结构和算法来管理进程生命周期的。本章主要针对实验前的理论学习（任务 1）和源码机制分析（任务 2）进行阐述。

2.1 任务 1：理解进程抽象与状态模型

进程是操作系统中最核心的抽象之一，它将一个静态的程序实例转化为一个动态的、拥有独立资源的执行实体。在 xv6 中，这一抽象主要通过进程控制块（PCB）和状态机来实现。

2.1.1 进程控制块结构分析

在 kernel/proc.h 中定义的 struct proc 是内核管理进程的核心数据结构 。通过分析源码可知，其关键字段主要包含以下几类：

- **标识与锁信息**：pid 是进程的唯一标识符；lock 是互斥自旋锁，用于保证在多核环境下对进程状态修改的原子性，防止竞争条件。
- 状态信息：state 字段记录了进程当前的生命周期状态（如 RUNNABLE, RUNNING 等）；killed 和 xstate 分别记录了进程是否收到终止信号及其退出码。
- 上下文保存区：这是实现多任务的关键。trapframe 用于保存用户态进入内核态时的寄存器现场（处理系统调用和中断）；而 context 则专门用于内核线程切换，保存了 callee-saved 寄存器（ra, sp, s0-s11），确保进程调度时能正确恢复内核栈和执行流 。
- 内存与栈：pagetable 指向进程独立的用户页表；kstack 记录了内核栈的虚拟地址，用于执行内核代码。

2．1．2 进程状态流转逻辑

进程的生命周期是一个有限状态机 。理解各状态间的转换条件是实现调度器的基础：

- UNUSED：进程槽位空闲，未被分配。
- USED：已分配槽位，正在进行初始化（如分配内存、页表）。
- RUNNABLE：进程已准备就绪，等待调度器分配 CPU 时间片。
- RUNNING：进程当前正在 CPU 上执行。
- SLEEPING：进程因等待 I/O 或其他事件（如 wait）而主动放弃 CPU。
- ZOMBIE：进程已退出，但其父进程尚未调用 wait 回收其资源。此时进程主体已销毁，仅保留 PCB 供父进程查询状态，这是防止内存泄漏和维护父子关系的重要机制 。

图 1 进程状态流转逻辑图

2.2 任务 2：分析 xv6 的进程创建机制

xv6 的进程创建主要依赖 allocproc 和 fork 两个核心函数。前者负责“无中生有”地构造一个新进程的基础环境，后者则负责“克隆”父进程的资源。

2．2．1 进程分配与初始化（allocproc）

allocproc 的职责是在全局进程数组中寻找一个空闲槽位并进行初始化 。其核心流程如下：

1. 查找空闲槽：遍历 proc 数组，寻找状态为 UNUSED 的结构体，加锁后将其状态修改为 USED，防止其他 CPU 抢占。
2. 分配 PID：利用全局计数器为新进程分配唯一的进程号。
3. 构建上下文：这是最关键的一步。函数会设置 context.ra 指向 forkret

函数地址，设置 context.sp 指向内核栈顶。这意味着当该进程第一次被调度器调度（swtch）时，CPU 将跳转到 forkret，并从刚设置好的内核栈开始执行，从而巧妙地启动进程 。设置特权级:将 mstatus 的 MPP 位设置为 Supervisor，确保执行 mret 后切换到 S 模式。

2.2.2 资源复制与 Fork 机制

sys_fork 是 Unix/Linux 系统创建进程的标准方式。在 xv6 中，它的实现逻辑极其严谨 ：

1. 镜像拷贝：调用 uvmcopy 将父进程的用户内存空间（代码、数据、栈）完整复制给子进程，确保两者运行环境一致，但物理内存隔离。

2. 继承 Trapframe：将父进程的 trapframe 内容逐字节拷贝给子进程。这保证了子进程开始运行时,其寄存器状态(包括 PC 指针)与父进程调用 fork 瞬间完全一致。

3. 区分返回值：为了让父子进程区分彼此，代码显式将子进程 trapframe->a0（返回值寄存器）修改为 0，而父进程则返回子进程的 PID 。

4. 构建文件视图：增加父进程打开文件的引用计数，使子进程共享相同的文件描述符表。

2.2.3 退出与回收机制

进程的结束涉及 exit 和 wait 的配合。当进程调用 exit 时，它会关闭打开的文件，将状态设为 ZOMBIE，并唤醒父进程。父进程在 wait 中负责捕获这一状态，释放子进程的内核栈和 PCB 槽位。这种机制确保了资源的有序回收，避免了“孤儿进程”和资源泄露的问题 。

3 实验原理与设计

3.1 总体架构设计

本实验旨在构建一个支持多任务并发的操作系统内核。系统设计遵循“分时复用”的原则，通过将物理 CPU 的计算资源在时间上切片，虚拟化为多个逻辑上的执行流，即“进程”。总体架构主要由以下三个核心模块组成：

1. 进程管理模块：负责 PCB 的分配与回收，维护进程从创建、运行到销毁

的完整生命周期。

2. 上下文切换模块：位于汇编层，负责在不同进程的内核栈之间切换 CPU 的寄存器状态，实现执行流的挂起与恢复。

3. 调度器模块：基于 RR 策略，公平地选择就绪进程占用 CPU，并处理时钟中断触发的抢占逻辑。

3.2 PCB 设计

为进程控制块是操作系统感知进程存在的唯一实体。在 include/proc.h 中，定义了 struct proc 结构体来描述一个进程的所有属性。为了支持后续的调度与管理，PCB 的设计包含以下关键字段：

- 标识符与状态：包含 pid 和 state（当前状态）。其中，状态 state 采用枚举类型 enum procstate 定义，精确描述了进程的生命周期，包括 UNUSED（空闲）、USED（分配未初始化）、RUNNABLE（就绪）、RUNNING（运行）、SLEEPING（睡眠）和 ZOMBIE（僵尸）六种状态。
- 上下文保存区：定义了 struct context 结构体，用于保存内核线程切换时的寄存器状态（包括 ra, sp, s0-s11）。这是实现并发的核心，确保进程被挂起后能准确恢复执行现场。
- 内核栈：kstack 字段记录了该进程独享的内核栈基地址。每个进程在创建时都会分配 4KB (KSTACK_SIZE) 的物理页作为栈空间，确保内核代码执行时的栈隔离。
- 同步与亲缘关系：chan 字段作为睡眠等待的“通道”，用于实现条件变量机制；parent 指针用于维护父子进程树，配合 wait_proc 实现资源回收。

```
struct proc {
    struct spinlock lock;
    enum procstate state;
    int pid;
    struct proc *parent;
    void *chan;
    int killed;
    int xstate;
    uint64 sz;                  // [新增] 进程的内存大小 (size)
    uint64 kstack;              // 内核栈
    struct trapframe *trapframe; // 新增: 指向 trapframe 物理页
    pagetable_t pagetable;      // 用户页表
    struct context context;     // 进程上下文
    void (*entry)(void);
    char name[16];
    struct file *ofile[NOFILE];
    struct inode *cwd;
};
```

图 2 进程数据结构定义

3.3 上下文切换机制设计

上下文切换是多任务系统的基石。本实验的设计重点在于区分“调用者保存”与“被调用者保存”寄存器。

由于 xv6 的调度过程本质上是一次函数调用（swtch），根据 RISC-V 的调用约定，只有 Callee-saved 寄存器需要在函数内部被显式保存。因此，在 kernel/swtch.S 中，我设计了如下切换逻辑：

1. 保存旧现场：将当前的 ra（返回地址）、sp（栈指针）以及 s0-s11（通用寄存器）保存到当前进程的 context 结构中。

2. 切换新现场：接收目标进程的 context 指针，从中恢复上述寄存器的值。

3. 栈切换：随着 sp 寄存器的更新，CPU 的栈环境瞬间从“旧进程内核栈”切换到了“新进程内核栈”。当 swtch 函数执行 ret 指令时，程序计数器（PC）将跳转到新进程上次被挂起的位置（即 context.ra 指向的地址），从而完成控制流的转移。

```
# kernel/swtch.S 核心逻辑
sd ra, 0(a0)    # 保存返回地址
sd sp, 8(a0)    # 保存栈指针
...
ld ra, 0(a1)    # 恢复返回地址
ld sp, 8(a1)    # 恢复栈指针
ret             # 跳转回新进程代码流
```

图 3 上下文切换核心逻辑

3.4 调度策略与状态机设计

3.4.1 进程状态流转

系统通过严格的状态机模型管理进程。设计中需要特别关注以下转换路径：

- 创建：alloc_proc 分配槽位（UNUSED -> USED），create_kthread 完成初始化（USED -> RUNNABLE）。
- 调度：调度器选中进程（RUNNABLE -> RUNNING）。
- 抢占：时钟中断触发 yield（RUNNING -> RUNNABLE）。
- 阻塞：进程等待 I/O 或锁（RUNNING -> SLEEPING）。
- 退出：进程结束或被杀（RUNNING/SLEEPING -> ZOMBIE）。

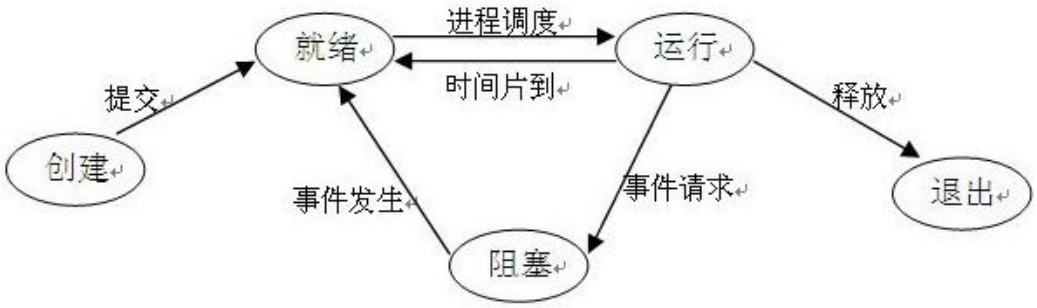


图 4 进程状态转换图

3．4．2 时间片轮转调度器

针对调度器采用无限循环的轮询机制，实现在 kernel/proc.c 的 scheduler 函数中。

1. 机制:CPU 结构体维护一个 scheduler 线程。该线程不断遍历全局 proc 数组，寻找状态为 RUNNABLE 的进程。

2. 策略：一旦找到就绪进程，将其状态修改为 RUNNING，并将 CPU 的上下文切换到该进程的上下文。

3. 抢占：通过 Lab 4 实现的时钟中断机制，当定时器触发时，内核调用 yield() 函数。该函数将当前进程状态改回 RUNNABLE，并调用 sched() 再次切换回 scheduler 线程，从而实现了基于时间片的抢占式多任务。

3.5 进程同步机制设计

为了协调进程间的执行顺序，设计了基于“睡眠通道”的同步机制：

- Sleep：当进程需要等待特定事件（如等待子进程退出）时，调用 sleep(chan)。该函数原子性地将进程状态设为 SLEEPING，记录等待通道 chan，并主动调用 sched() 让出 CPU。
- Wakeup ：当事件发生时（如子进程调用 exit），相关内核代码调用 wakeup(chan)。该函数遍历进程表，查找所有等待在 chan 上的睡眠进程，并将其状态重置为 RUNNABLE，使其能够被调度器再次选中。

这种设计避免了低效的“忙等待”，有效提升了 CPU 利用率。

4 实验步骤与实现

4.1 进程管理系统的初始化

系统启动初期，内存中充斥着随机数据。为了构建可靠的进程环境，首要任务是在 kernel/proc.c 中实现 proc_init 函数，对全局进程表进行“清零”操作。

- 遍历初始化：使用循环遍历全局数组 struct proc proc[NPROC]。
- 状态重置：将所有进程槽位的状态强制设为 UNUSED，并将 kstack（内核栈指针）、pid 等关键字段清零。这确保了后续分配进程时不会误用脏数据。
- CPU 初始化：同时初始化 cpus 结构体，清空当前运行进程指针，为调度器的启动做好准备。

4.2 PCB 的分配与构造

进程的创建始于 PCB 的分配。在 alloc_proc 函数中，实现了从“无”到“有”的构造过程：

1. 寻找空闲槽位：遍历进程表，寻找状态为 UNUSED 的槽位。若找不到，则返回失败。

2. 分配内核资源：

   - PID 分配：使用原子递增的全局变量 nextpid 为新进程赋予唯一标识。

   - 内核栈分配：调用内存管理模块的 alloc_page() 分配 4KB 物理页作为内核栈，并将栈底地址记录在 p->kstack 中。

3. 上下文预设：这是最关键的一步。通过 memset 清空 context 结构，并根据 RISC-V 调用约定初始化关键寄存器：

   - context.ra（返回地址）：设置为线程入口函数的地址。

   - context.sp（栈指针）：设置为 p->kstack + KSTACK_SIZE，即栈顶位置。

   - 逻辑意义：这种构造“伪造”了一个仿佛刚从该函数被挂起的现场。当调度器第一次切换到该进程时，CPU 执行 ret 指令，就会直接跳转到入

口函数开始执行，实现了从静态数据到动态执行流的转换。

4.3 上下文切换的汇编实现

为了实现内核线程间的切换，我在 kernel/swtch.S 中编写了核心汇编函数 swtch(struct context *old, struct context *new)。由于 C 语言无法直接操作 PC 指针和栈指针的原子切换，该部分必须使用汇编实现。

1. 保存现场：接收 a0 寄存器（指向旧进程 context），依次执行 sd 指令，将 ra, sp 以及 Callee-saved 通用寄存器（s0-s11）保存到旧进程的上下文结构体中。

2. 恢复现场：接收 a1 寄存器（指向新进程 context），依次执行 ld 指令，将上述寄存器的值从新进程的上下文加载到 CPU 物理寄存器中。

3. 原子跳转：最后执行 ret 指令。此时 ra 寄存器已被更新为新进程的返回地址，CPU 将跳转到新进程上次停止的代码位置继续执行。

4.4 抢占式调度器的全链路实现

调度器是操作系统的“心脏”。在 kernel/proc.c 中，我实现了一个基于状态轮询的调度器 scheduler，并配合时钟中断实现了抢占机制。

1. 调度循环：每个 CPU 核心运行一个死循环，不断检查进程表。

   - 开启中断：在选取进程前，通过 w_sstatus 开启全局中断（SIE），确保在空闲等待时能响应外部事件（如键盘或时钟）。
   - 寻找就绪进程：遍历数组，寻找 state == RUNNABLE 的进程。

2. 执行切换：一旦找到目标进程：

   - 将进程状态修改为 RUNNING。
   - 将 CPU 的当前进程指针 c->proc 指向该进程。
   - 调用 swtch(&c->context, &p->context)，将控制权移交给新进程。

3. 主动/被动让出：

   - 被动抢占：当 Timer 中断触发时，中断处理程序调用 yield()。该函数将状态改回 RUNNABLE，并调用 sched() 切回调度器上下文。
   - 主动放弃：当进程需要等待资源（如 sleep）时，也会主动调用 sched()，从而避免忙等待。

```c
void scheduler(void) {
    struct cpu *c = mycpu();
    c->proc = 0;
    printf("scheduler: starting on cpu 0\n");
    while(1) {
        intr_on();
        for (struct proc *p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if (p->state == RUNNABLE) {
                p->state = RUNNING;
                c->proc = p;
                swtch(&c->context, &p->context);
                c->proc = 0;
            }
            release(&p->lock);
        }
    }
}
```

图 5 scheduler 调度流程代码逻辑

4.5 进程退出与资源回收

在完整的进程生命周期必须包含“死亡”与“回收”。本实验设计了 exit、kill 与 wait 机制来处理这一流程。

- 进程终止 (exit_proc)：当进程结束时：

  1. 处理子进程：调用 kill_children 递归标记并唤醒所有子进程，防止它们成为无主的“孤儿进程”还在后台运行。
  2. 状态变更：将自身状态设为 ZOMBIE，保存退出码 xstate。
  3. 唤醒父进程：调用 wakeup(p->parent)，通知父进程来收尸。
  4. 最后一次切换：调用 sched() 进入调度器，且永远不再返回。

- 父进程回收 (wait_proc)：父进程通过该函数查找状态为 ZOMBIE 的子进程。

  1. 若找到，则复制其 PID 和退出码，调用 free_proc 释放其内核栈和 PCB 槽位，彻底清除其痕迹。
  2. 若子进程还在运行，父进程则调用 sleep 进入睡眠状态，等待被唤醒。

- 递归清理 (kill_proc)：针对复杂的进程树结构，实现了递归查杀功能。当杀死一个进程时，系统会深度遍历进程表，将该进程及其所有后代进程的 killed 标志置位，并强制唤醒睡眠中的进程，确保整个进程组能被快速终止。

4.6 辅助调试与监控接口

为了验证系统的正确性并直观展示调度行为，实现了 proc_info 和 proc_stats 函数。它们遍历进程表，格式化打印每个进程的 PID、状态、名称以及运行时间。同时，系统维护了一个全局计数器 total_switches，在每次 sched() 调用时自增，用于量化分析调度器的切换频率和负载情况。

5 实验测试与结果

本章将展示进程管理系统的实际运行效果。测试代码集成在 kernel/test.c 中，由 main_task 统一调用 run_lab5_tests() 执行。测试涵盖了进程创建极限、调度器抢占以及进程间同步三个关键场景。

5.1 进程创建与资源限制测试

首先验证内核对进程数量上限的管理以及 alloc_proc 的资源分配能力。

- **测试方法：**调用 test_process_creation() 函数。该测试在一个循环中不断调用 create_process(simple_task)，直到返回失败。目的是验证系统能否正确处理 NPROC（最大进程数，通常为 64）的边界情况，并在测试结束后通过 wait_process 回收所有资源。
- **测试结果：**如图 6 所示，控制台输出 Created 62 processes (expected max 62)。随后测试程序成功回收了所有子进程，并输出 Process creation test passed。
- **结果分析：**实验结果证明了系统能够稳定支持多进程并发创建，且 alloc_proc 函数正确实现了对进程表（Process Table）满员情况的检测与拒绝（返回 0 或 -1），防止了内核因资源耗尽而崩溃。

```
===== Starting Lab5 Tests =====

=== Test 5.1: Process Creation ===
Testing basic process creation...
Created process 2
Existing process: PID=1, state=4, name=(unnamed)
Existing process: PID=2, state=3, name=(unnamed)
Available process slots: 62

Testing process table limit (NPROC=64)...
Created 62 processes (expected max 62)
```

图 6 进程创建测试结果输出

5.2 抢占式调度器测试

本节重点验证基于时间片轮转的调度策略及定时器中断驱动的抢占机制。

- **测试方法：**调用 test_scheduler() 函数。主线程创建 3 个 cpu_intensive_task（CPU 密集型任务，不主动让出 CPU），随后主线程调用 kernel_sleep(1000) 进入睡眠。
- **测试结果：**如图 7 所示，控制台首先打印 Creating 3 CPU intensive tasks...。在随后的运行中，尽管这些任务是死循环计算，但控制台依然显示主线程在睡眠结束后成功被唤醒，并打印 Scheduler test completed (slept 1000 ticks)。若开启了任务内部打印，还能看到三个任务交替执行的日志。
- **结果分析：**该结果有力地证明了时钟中断抢占机制的有效性。即使用户任务陷入死循环，硬件定时器也能周期性触发中断，强制当前进程调用 yield() 让出 CPU，从而保证了 scheduler 能够重新获得控制权并调度其他进程（包括唤醒睡眠中的主线程），实现了真正的并发。

```
=== Test 5.2: Scheduler ===
Creating 3 CPU intensive tasks...
Observing scheduler behavior (sleeping 1000 ticks)...
PID 65: cpu_intensive_task running...
PID 65: cpu_intensive_task finished.
PID 66: cpu_intensive_task running...
PID 66: cpu_intensive_task finished.
PID 67: cpu_intensive_task running...
PID 67: cpu_intensive_task finished.
Scheduler test completed (slept 1000 ticks)
```

图 7 调度器测试结果输出

5.3 进程同步与通信测试

最后验证基于条件变量（Sleep/Wakeup）的进程同步机制。

- **测试方法：**调用 test_synchronization() 函数，模拟经典的“生产者-消费者”模型。
  - 初始化共享缓冲区 shared_buffer_init()。
  - 创建 producer_task：向缓冲区写入数据，满时睡眠。
  - 创建 consumer_task：从缓冲区读取数据，空时睡眠。
- **测试结果：**如图 8 所示，控制台输出了有序的生产与消费日志。例如

Producer produced item 10 后，缓冲区满，生产者暂停；随后 Consumer consumed item 10，缓冲区腾出空间，生产者继续运行。最终输出 Synchronization test passed。

- **结果分析：**测试表明 sleep() 和 wakeup() 机制工作正常。当缓冲区满或空时，进程能够正确地将状态切换为 SLEEPING 并让出 CPU；而当条件满足时，另一方能准确地唤醒等待队列中的进程（RUNNABLE）。这证实了内核实现了正确的进程间通信（IPC）和临界区保护。

```
=== Test 5.3: Synchronization (Pr
PID 68: producer_task running...
Producer: produced 0 (count=1)
Producer: produced 1 (count=2)
Producer: produced 2 (count=3)
Producer: produced 3 (count=4)
Producer: produced 4 (count=5)
Producer: produced 5 (count=6)
Producer: produced 6 (count=7)
Producer: produced 7 (count=8)
Producer: produced 8 (count=9)
Producer: produced 9 (count=10)
PID 69: consumer_task running...
Consumer: consumed 9 (count=9)
Consumer: consumed 8 (count=8)
Consumer: consumed 7 (count=7)
Consumer: consumed 6 (count=6)
Consumer: consumed 5 (count=5)
Consumer: consumed 4 (count=4)
Consumer: consumed 3 (count=3)
Consumer: consumed 2 (count=2)
Consumer: consumed 1 (count=1)
Consumer: consumed 0 (count=0)
Producer: produced 10 (count=1)
Producer: produced 11 (count=2)
Producer: produced 12 (count=3)
Producer: produced 13 (count=4)
Producer: produced 14 (count=5)
Producer: produced 15 (count=6)
Producer: produced 16 (count=7)
Producer: produced 17 (count=8)
Producer: produced 18 (count=9)
Producer: produced 19 (count=10)
Producer finished.
Consumer: consumed 19 (count=9)
Consumer: consumed 18 (count=8)
Consumer: consumed 17 (count=7)
Consumer: consumed 16 (count=6)
Consumer: consumed 15 (count=5)
Consumer: consumed 14 (count=4)
Consumer: consumed 13 (count=3)
Consumer: consumed 12 (count=2)
Consumer: consumed 11 (count=1)
Consumer: consumed 10 (count=0)
Consumer finished.
Synchronization test completed
```

图 8 同步机制测试结果输出

6 遇到的问题及解决

6.1 问题 1：上下文切换后系统崩溃

- **问题描述：**在完成 proc.c 和 swtch.S 后尝试运行第一个内核线程。系统打印 "Scheduler started" 后立即崩溃，QEMU 报错提示跳转到了非法的内存地址（具体是 0x00000000），导致取指异常。
- **原因分析：**经过 GDB 单步调试跟踪，发现问题出在 alloc_proc 函数的上下文初始化环节。虽然我分配了内核栈并清空了 context 结构体，但

没有正确设置 context.ra。在 RISC-V 的 ret 指令执行时，CPU 会将 ra 寄存器的值加载到 PC。由于 ra 被初始化为 0，swtch 返回后 CPU 试图跳转到地址 0 执行代码，从而触发了异常。

- **解决方案：**在 alloc_proc 函数中显式初始化 ra 和 sp。将 p->context.ra 指向线程的入口函数地址，并将 p->context.sp 指向分配好的内核栈顶（p->kstack + KSTACK_SIZE）。修改后，swtch 能够正确跳转到目标函数执行。

6.2 问题 2：时钟中断无法抢占死循环进程

- **问题描述：**在进行调度器测试时，我创建了一个执行死循环的计算密集型任务。预期系统应通过时钟中断强制切换回调度器，但实际上该任务一旦开始运行，就独占了 CPU，控制台不再输出任何其他日志，系统仿佛“卡死”。
- **原因分析：**检查 scheduler 函数发现，调度器在选择下一个进程并调用 swtch 之前，没有显式开启全局中断。在 RISC-V 中，中断默认在异常处理入口被硬件自动关闭。如果 scheduler 运行在关中断状态下切换到新进程，而新进程中又没有主动开启中断，那么 sstatus.SIE 位始终为 0。这意味着时钟中断信号被 CPU 屏蔽，无法触发 timervec 中的 yield 调用，导致抢占失效。
- **解决方案：**在 scheduler 函数的主循环中，在遍历进程表之前，添加 w_sstatus(r_sstatus() | SSTATUS_SIE); 指令，确保在 CPU 空闲或寻找进程时处于“开中断”状态。同时，确保进程在初始化时，其 sstatus 的 SPIE 位被置 1，这样在中断返回进入进程执行时，硬件会自动开启中断。

7 思考题

7.1 进程模型

7.1.1 为什么选择这种进程结构设计？

xv6 的 struct proc 设计采用了“扁平化”和“静态数组”的策略。

- **简单可靠：**使用静态数组 proc[NPROC] 避免了内核态复杂的动态内存分配，减少了内存碎片和由于内存不足导致内核崩溃的风险。
- **功能完备：**结构体聚合了调度（context/state）、内存（pagetable）、亲缘关系（parent）和文件系统（ofile）等所有必要信息，符合单体内核（Monolithic Kernel）高内聚的设计思想。

7.1.2 如何支持轻量级线程？

当前的进程模型中，每个进程拥有独立的页表。要支持轻量级线程，可以在 struct proc 中通过 clone 系统调用实现资源共享：

- **共享页表：**创建新线程时，不复制页表，而是让新线程的 pagetable 指针指向父进程的页表。
- **独立栈：**虽然共享地址空间，但必须为每个线程分配独立的 trapframe 和用户栈/内核栈，以保证执行流独立。

7.2 调度策略

7.2.1 轮转调度的公平性如何？

本次实验实现的时间片轮转算法在理论上具有较好的公平性。它保证了所有就绪状态的进程都能轮流获得 CPU 时间片（如 100ms），不会出现某个进程长期饥饿的情况。上下文切换带来的内存访问延迟，特别是保存和恢复 32 个通用寄存器及 CSR 状态时会产生大量的内存读写，且容易引发 Cache Miss。

但仍存在**不足**：它对所有进程一视同仁，无法区分 I/O 密集型和 CPU 密集型任务。I/O 密集型任务（如键盘输入）需要快速响应，但可能会被排在长作业后面等待，导致响应延迟。

7.2.1 如何实现实时调度？

要满足实时性，需要引入优先级和抢占时限：

- **优先级调度：**在 struct proc 中增加 priority 字段。调度器不再简单遍历，而是始终选择优先级最高的 RUNNABLE 进程。

- **最早截止时间优先（EDF）：**对于硬实时任务，根据任务的 Deadline 动态调整优先级，越快过期的任务优先级越高。

7.3 性能优化

7.3.1 fork() 的性能瓶颈在哪里？如何解决？

- **瓶颈：**sys_fork 最大的开销在于 uvmcopy，即物理内存的完整拷贝。当父进程占用大量内存时，逐页复制数据非常耗时且浪费内存。
- **优化：**实现 写时复制（Copy-On-Write，COW） 技术。fork 时只复制页表项，不复制物理页，并将父子进程的页表项都设为“只读”。当任一方尝试写入时，触发缺页异常，内核捕获异常后，才真正分配物理页并复制数据，从而大幅降低 fork 的延迟。

7.3.2 上下文切换开销如何降低？

- **减少寄存器保存：**利用编译器优化，仅保存必要的 Callee-saved 寄存器（如实验中 swtch.S 所做）。
- **硬件加速：**某些架构提供多组影子寄存器，切换时只需改变硬件指针。
- **避免 TLB 刷新：**在支持 ASID（地址空间 ID）的 CPU 上，切换页表时不需要清空 TLB，从而减少切换后的缓存未命中开销。

7.4 资源管理

7.4.1 如何实现进程资源限制？

可以在 struct proc 中增加资源计数器和限制阈值字段。在系统调用入口（如 sys_sbrk, sys_fork, sys_open）检查当前累积值是否超过阈值（如最大内存使用量、最大打开文件数）。若超过，则拒绝请求并返回错误。

7.4.2 如何处理进程资源泄漏？

- 引用计数：对共享资源（如文件、物理页）使用引用计数，只有当计数为 0 时才物理释放。
- 父进程托底：如实验实现的 wait 机制，父进程负责回收子进程的 PCB 和内核栈。
- Init 进程接管：如果父进程先于子进程退出，必须将子进程“过继”给

系统的第一个进程（Init），由 Init 进程循环调用 wait 来回收这些孤儿进程，防止僵尸进程永久占用进程表。

8 实验总结

通过本次实验，我成功在基于 RISC-V 的内核中构建了一个功能完备的进程管理子系统，完成了从单一执行流向多任务并发环境的跨越。实验初期，我首先着眼于进程抽象的构建，通过设计包含状态、内核栈及上下文信息的 PCB，明确了操作系统管理计算资源的基本单元。在实现核心的上下文切换机制时，我深入研读了 RISC-V 的函数调用规范，通过编写 swtch.S 汇编代码，精确控制了 ra、sp 及通用寄存器的保存与恢复，这一过程让我深刻理解了“线程”本质上就是一组被冻结的寄存器状态，而“切换”则是 CPU 视角的瞬间场景变换。

实验中最具挑战性也最具启发性的环节是抢占式调度器的实现。我将 Lab 4 中实现的时钟中断与本次实验的调度策略有机结合，通过在中断处理程序中强制调用 yield，打破了程序独占 CPU 的局面，实现了真正意义上的时间片轮转。这种软硬件协同的机制让我直观体会到了操作系统如何作为“幕后黑手”公平地分配 CPU 时间。此外，针对进程间协作问题，我设计了基于 sleep 和 wakeup 的同步原语，并通过经典的生产者-消费者模型验证了其正确性，解决了多进程环境下的竞态条件与资源同步难题。

最后，通过实现完整的进程生命周期管理——从 fork 的资源克隆到 exit 的僵尸态转化，再到 wait 的最终回收，我构建了一个闭环的资源管理系统。本次实验不仅验证了课堂上关于进程状态机和调度算法的理论，更让我从代码层面掌握了并发系统的核心奥秘。当看到多个内核线程在控制台上交替打印输出时，我真切感受到了操作系统赋予计算机的“同时做多件事”的生命力，这也为未来引入用户态进程和更复杂的虚拟内存管理奠定了坚实的基石。

教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）