Lab5

编程题

1. 实现一个使用 nice,fork,exec,spawn 等与进程管理相关的系统调用的 linux 应用程序。

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
int childpid;  // 存储子进程的进程ID
int i;
// 未使用的整型变量i
if (fork() == 0)  // 子进程
{
char * execv_str[] = {"echo", "child process, executed by execv",NULL};
// 定义一个字符串数组，用于存储要通过execv()执行的命令和参数
if (execv("/usr/bin/echo",execv_str) <0 ){  // 使用execv()函数执行指定的命令（这里是echo），若返回值小于0表示执行出错
perror("error on exec\n");  // 打印错误信息
exit(0);  // 退出子进程
}
}
else  // 父进程
{
wait(&childpid);  // 父进程等待子进程结束，使用wait()函数，并通过&childpid参数
//存储子进程的退出状态
printf("parent process, execv done\n");  // 子进程执行完毕后，父进程打印消息
}
return 0;
}
```

如图所示建立程序，运行程序得到结果：

```
oslab@oslab-virtual-machine:~/lab5/rCore-sp23$ ./1
child process, executed by execv
parent process, execv done
```

sys_spawn 这个系统调用的主要功能就是在操作系统内核中“创建并启动”一个新的用户进程，类似用户态的 fork+exec 组合。可以借鉴这二者的写法， path 部分获取就是模仿 exec 写的。 实现该系统调用的主要想法是给新的应用数据 elf_data 建立一个新的进程模块， 这意味着我们需要以 exec 的角度去改编 fork。一直到 TrapContext 这部分， 前面的内容和 fork 基本一致， 但是我们需要运行一个新的应用， 因而不能像 fork 那样沿用与其父进程一致的配置， 而需要创建一个全新的 TrapContext 以容纳 entry_point， user_sp， kernel_stack_top 这几个依托当前应用数据生成的信息。 最后也是最关键的， 我们需要返回子进程的 pid， 并且将这个子进程的进程控制模块加入到 task 队列中， 以使得操作系统能够分配资源执行该应用。

2. 请阅读下列代码，分析程序的输出 A 的数量：（已知 && 的优先级比 || 高）
int main() { fork() && fork() && fork() || fork() && fork() || fork() && fork(); printf("A"); return 0; } 如果给出一个 && || 的序列，如何设计一个程序来得到答案？

```python
class Node:
    def __init__(self, typ, left=None, right=None, fork_index=None):
        self.typ = typ            # 'fork', 'and', 'or'
        self.left = left          # 左子节点
        self.right = right        # 右子节点
        self.fork_index = fork_index  # fork() 的编号（按出现顺序）

class Parser:
    def __init__(self, s):
        self.s = s
        self.i = 0
        self.n = len(s)
        self.fork_count = 0

    def skip_spaces(self):
        while self.i < self.n and self.s[self.i].isspace():
            self.i += 1

    def parse(self):
        node = self.parse_or()
        self.skip_spaces()
        if self.i < self.n:
            raise Exception("语法错误：表达式未能完整解析")
        return node

    def parse_or(self):
        left = self.parse_and()
        self.skip_spaces()
        # 处理 || 运算
        while self.i+1 < self.n and self.s[self.i] == '|' and self.s[self.i+1] == '|':
            self.i += 2
            right = self.parse_and()
            left = Node('or', left, right)
            self.skip_spaces()
        return left

    def parse_and(self):
        left = self.parse_term()
        self.skip_spaces()
        # 处理 && 运算
        while self.i+1 < self.n and self.s[self.i] == '&' and self.s[self.i+1] == '&':
            self.i += 2
            right = self.parse_term()
            left = Node('and', left, right)
            self.skip_spaces()
        return left
```

```
    def parse_term(self):
        self.skip_spaces()
        # 处理括号
        if self.i < self.n and self.s[self.i] == '(':
            self.i += 1
            node = self.parse_or()
            self.skip_spaces()
            if self.i < self.n and self.s[self.i] == ')':
                self.i += 1
                return node
            else:
                raise Exception("缺少右括号")
        # 处理 fork()
        if self.i+5 <= self.n and self.s[self.i:self.i+5] == "fork(":
            if self.i+6 <= self.n and self.s[self.i+5] == ')' and self.s[self.i:self.i+6] == "fork()":
                self.i += 6
                self.fork_count += 1
                return Node('fork', None, None, fork_index=self.fork_count)
        raise Exception(f"语法错误：在索引 {self.i} 处发现意外符号 '{self.s[self.i:]}'")

class Context:
    """表示一个进程分支的上下文，path 存储经过的 fork 返回情况。"""
    def __init__(self, path=None, value=None):
        self.path = path or []  # 列表形式记录 (fork_index, Bool) 的返回值
        self.value = value      # 当前子表达式的布尔值

    def copy(self):
        return Context(path=self.path.copy(), value=self.value)
```

```python
def simulate_node(node, contexts):
    """
    递归模拟节点，contexts 是当前进程列表（Context 对象）。
    返回新的进程上下文列表。
    """
    if node.typ == 'fork':
        newcontexts = []
        for c in contexts:
            # 父进程分支（返回 True，非0）
            c1 = c.copy()
            c1.path.append((node.fork_index, True))
            c1.value = True
            # 子进程分支（返回 False，0）
            c2 = c.copy()
            c2.path.append((node.fork_index, False))
            c2.value = False
            newcontexts.extend([c1, c2])
        return newcontexts

    elif node.typ == 'and':
        out = []
        for c in contexts:
            # 先计算左子表达式
            left_results = simulate_node(node.left, [Context(path=c.path.copy())])
            for lr in left_results:
                # 左真则继续计算右子表达式
                if lr.value:
                    right_results = simulate_node(node.right, [Context(path=lr.path.copy())])
                    out.extend(right_results)
                else:
                    # 左假则跳过右边，整体为假
                    out.append(Context(path=lr.path.copy(), value=False))
        return out

    elif node.typ == 'or':
        out = []
        for c in contexts:
            left_results = simulate_node(node.left, [Context(path=c.path.copy())])
            for lr in left_results:
                if lr.value:
                    # 左真则整体为真（跳过右边）
                    out.append(Context(path=lr.path.copy(), value=True))
                else:
                    # 左假则计算右边
                    right_results = simulate_node(node.right, [Context(path=lr.path.copy())])
                    out.extend(right_results)
        return out
```

```python
def simulate_expression(expr):
    parser = Parser(expr)
    ast = parser.parse()
    # 从初始上下文开始模拟
    contexts = simulate_node(ast, [Context(path=[], value=None)])
    return contexts, parser.fork_count

def print_results(expr):
    contexts, total_forks = simulate_expression(expr)
    # 每个进程分支打印 A，故打印总次数即为分支数
    print(f"总打印次数: {len(contexts)}")
    for i, c in enumerate(contexts, 1):
        parts = []
        executed = set(idx for idx, _ in c.path)
        # 列出每个 fork() 返回情况
        for idx, val in c.path:
            parts.append(f"fork{idx}={'父(非0)' if val else '子(0)'}")
        # 列出被短路跳过的 fork()
        skipped = [f"fork{idx}=跳过(短路)" for idx in range(1, total_forks+1) if idx not in executed]
        desc = ', '.join(parts + skipped)
        print(f"进程{i}: {desc}, 打印A")

# 示例运行
expr = "fork() && fork() && fork() || fork() && fork() || fork() && fork()"
print_results(expr)
```

设计程序如上图所示，将题目中的字符串作为示例加入代码中，再运行程序，得到结果：

```
oslab@oslab-virtual-machine:~/lab5/rCore-sp23$ python3 2.py
总打印次数: 22
进程1: fork1=父(非0), fork2=父(非0), fork3=父(非0), fork4=跳过(短路), fork5=跳过(短路), fork6=跳过(短路), fork7=跳过(短路), 打印A
进程2: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=父(非0), fork5=父(非0), fork6=跳过(短路), fork7=跳过(短路), 打印A
进程3: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=父(非0), fork5=子(0), fork6=父(非0), fork7=父(非0), 打印A
进程4: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=父(非0), fork5=子(0), fork6=父(非0), fork7=子(0), 打印A
进程5: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=父(非0), fork5=子(0), fork6=子(0), fork7=跳过(短路), 打印A
进程6: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=子(0), fork6=父(非0), fork7=父(非0), fork5=跳过(短路), 打印A
进程7: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=子(0), fork6=父(非0), fork7=子(0), fork5=跳过(短路), 打印A
进程8: fork1=父(非0), fork2=父(非0), fork3=子(0), fork4=子(0), fork6=子(0), fork
```

总共有 22 个输出 A 的数量。

问答题

1 * 如何查看 Linux 操作系统中的进程?
使用 ps 命令，常用方法：$ ps aux
2 * 简单描述一下进程的地址空间中有哪些数据和代码。
代码(text)段，数据(data)段： 已初始化的全局变量的内存映射，bss 段：未初始化或默认初始化为 0 的全局变量，堆(heap)，用户栈(stack)，共享内存段

3 * 进程控制块保存哪些内容?
进程标识符、进程调度信息（进程状态，进程的优先级，进程调度所需的其它信息）、进程间通信信息、内存管理信息（基地址、页表或段表等存储空间结构）、进程所用资源（ I/O 设备列表、打开文件列表等）、处理机信息（通用寄存器、指令计数器、用户的栈指针）

4 * 进程上下文切换需要保存哪些内容?
页全局目录、部分寄存器、内核栈、当前运行位置

5 ** fork 为什么需要在父进程和子进程提供不同的返回值?
可以根据返回值区分父子进程，明确进程之间的关系，方便用户为不同进程执行不同的操作。

6 ** fork + exec 的一个比较大的问题是 fork 之后的内存页/文件等资源完全没有使用就废弃了，针对这一点，有什么改进策略?
采用 COW(copy on write),或使用使用vfork 等。

7 ** 其实使用了 6 的策略之后，fork + exec 所带来的无效资源的问题已经基本被解决了，

但是近年来 fork 还是在被不断的批判，那么到底是什么正在"杀死"fork？可以参考 论文 。
fork 和其他的操作不正交,也就是 os 每增加一个功能,都要改 fork，这导致新功能开发困难,设计受限.有些和硬件相关的甚至根本无法支持 fork.
fork 得到的父子进程可能产生共享资源的冲突；
子进程继承父进程，如果父进程处理不当，子进程可以找到父进程的安全漏洞进而威胁父进程；
还有比如 fork 必须要虚存, SAS 无法支持等等.

8 ** 请阅读下列代码，并分析程序的输出，假定不发生运行错误，不考虑行缓冲，不考虑中断：

```
int main(){
    int val = 2;

    printf("%d", 0);
    int pid = fork();
    if (pid == 0) {
        val++;
        printf("%d", val);
    } else {
        val--;
        printf("%d", val);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    return 0;
}
```

如果 fork() 之后主程序先运行, 则结果如何？如果 fork() 之后 child 先运行, 则结果如何？
答 01342 03412

9 ** 为什么子进程退出后需要父进程对它进行 wait，它才能被完全回收？

答 当一个进程通过 exit 系统调用退出之后，它所占用的资源并不能够立即全部回收，需要由该进程的父进程通过 wait 收集该进程的返回状态并回收掉它所占据的全部资源，防止子进程变为僵尸进程造成内存泄漏。同时父进程通过 wait 可以获取子进程执行结果，判断运行是否达到预期，进行管理。

10 ** 有哪些可能的时机导致进程切换？

答 进程主动放弃 cpu：运行结束、调用 yield/sleep 等、运行发生异常中断
进程被动失去 cpu：时间片用完、新进程到达、发生 I/O 中断等

11 ** 请描述在本章操作系统中实现本章提出的某一种调度算法（RR 调度除外）的简要实现步骤。

答 可降低优先级的 MLFQ：将 manager 的进程就绪队列变为数个，初始进程进入第一队列，调度器每次选择第一队列的队首进程执行，当一个进程用完时间片而未执行完，就在将它重新添加至就绪队列时添加到下一队列，直到进程位于底部队列。

12 * 非抢占式的调度算法，以及抢占式的调度算法，他们的优点各是什么？

答 非抢占式：中断响应性能好、进程执行连续，便于分析管理
抢占式：任务级响应时间最优，更能满足紧迫作业要求

13 ** 假设我们简单的将进程分为两种：前台交互（要求短时延）、后台计算（计算量大）。下列进程/或进程组分别是前台还是后台？a) make 编译 linux; b) vim 光标移动; c) firefox 下载影片; d) 某游戏处理玩家点击鼠标开枪; e) 播放交响乐歌曲; f) 转码一个电影视频。除此以外，想想你日常应用程序的运行，它们哪些是前台，哪些是后台的？

答 前台：b,d,e。后台：a,c,f

14 ** RR 算法的时间片长短对系统性能指标有什么影响？

答 时间片太大，可以让每个任务都在时间片内完成，但进程平均周转时间会比较长，极限情况下甚至退化为 FCFS;
时间片过小，反应迅速，响应时间会比较短，可以提高批量短任务的完成速度。但产生大量上下文切换开销，使进程的实际执行时间受到挤占。
因此需要在响应时间和进程切换开销之间进行权衡，合理设定时间片大小。

15 ** MLFQ 算法并不公平，恶意的用户程序可以愚弄 MLFQ 算法，大幅挤占其他进程的时间。（MLFQ 的规则：“如果一个进程，时间片用完了它还在执行用户计算，那么 MLFQ 下调它的优先级”）你能举出一个例子，使得你的用户程序能够挤占其他进程的时间吗？

答 每次连续执行只进行大半个时间片长度即通过执行一个 IO 操作等让出 cpu，这样优先级不会下降，仍能很快得到下一次调度。
16 *** 多核执行和调度引入了哪些新的问题和挑战？

答 多处理机之间的负载不均问题：在调度时，如何保证每一个处理机的就绪队列保证优先级、性能指标的同时负载均衡
数据在不同处理机之间的共享与同步问题：除了 Cache 一致性的问题，在不同处理机上同时运行的进程可能对共享的数据区域产生相同的数据要求，这时就需要避免数据冲突，采用同步互斥机制处理资源竞争;
线程化问题：如何将单个进程分为多线程放在多个处理机上
Cache 一致性问题：由于各个处理机有自己的私有 Cache，需要保证不同处理机下的 Cache 之中的数据一致性
处理器亲和性问题：在单一处理机上运行的进程可以利用 Cache 实现内存访问的优化与加速，这就需要我们规划调度策略，尽量使一个进程在它前一次运行过的同一个 CPU 上运行，也即满足处理器亲和性。

通信问题：类似同步问题，如何降低核间的通信代价

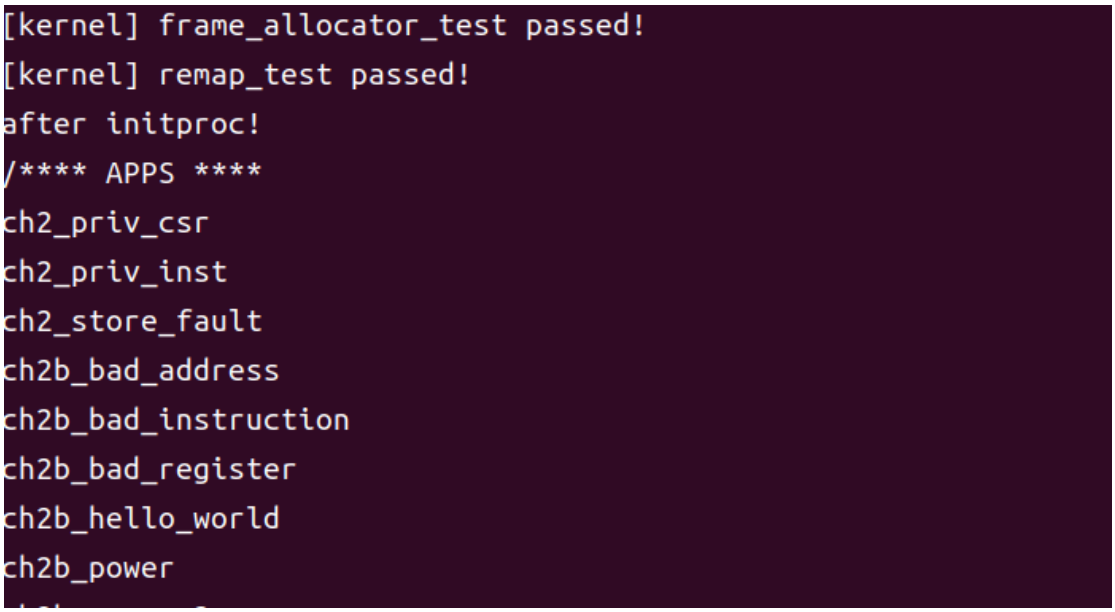
实验练习 1:

大家一定好奇过为啥进程创建要用 fork + execve 这么一个奇怪的系统调用，就不能直接搞一个新进程吗？思而不学则殆，我们就来试一试！这章的编程练习请大家实现一个完全 DIY 的系统调用 spawn，用以创建一个新进程。

sys_spawn 这个系统调用的主要功能就是在操作系统内核中“创建并启动”一个新的用户进程，类似用户态的 fork+exec 组合。可以借鉴这二者的写法， path 部分获取就是模仿 exec 写的。 实现该系统调用的主要想法是给新的应用数据 elf_data 建立一个新的进程模块， 这意味着我们需要以 exec 的角度去改编 fork。 一直到 TrapContext 这部分， 前面的内容和 fork 基本一致， 但是我们需要运行一个新的应用， 因而不能像 fork 那样沿用与其父进程一致的配置， 而需要创建一个全新的 TrapContext 以容纳 entry_point， user_sp， kernel_stack_top 这几个依托当前应用数据生成的信息。 最后也是最关键的， 我们需要返回子进程的 pid， 并且将这个子进程的进程控制模块加入到 task 队列中， 以使得操作系统能够分配资源执行该应用。

```rust
pub fn sys_spawn(path: *const u8) -> isize {
    let task = current_task().unwrap();
    let mut parent_inner = task.inner_exclusive_access();
    let token = parent_inner.memory_set.token();
    let path = translated_str(token, path);
    if let Some(elf_data) = get_app_data_by_name(path.as_str()) {
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT_BASE).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        let task_control_block = Arc::new(TaskControlBlock {
            pid: pid_handle,
            kernel_stack,
            inner: unsafe {
                UPSafeCell::new(TaskControlBlockInner {
                    task_status: TaskStatus::Ready,
                    task_cx: TaskContext::goto_trap_return(kernel_stack_top),
                    syscall_times: [0; MAX_SYSCALL_NUM],
                    user_time: 0,
                    kernel_time: 0,
                    checkpoint: get_time_ms(), // give the new process a new start point
                    memory_set,
                    trap_cx_ppn,
                    base_size: parent_inner.base_size,
                    heap_bottom: parent_inner.heap_bottom,
                    program_brk: parent_inner.program_brk,
                    parent: Some(Arc::downgrade(&task)),
                    children: Vec::new(),
                    exit_code: 0,
                    stride: 0,
                    priority: 16,
                })
            },
        });
        // add child
        parent_inner.children.push(task_control_block.clone());
        // prepare TrapContext in user space
        let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            kernel_stack_top,
            trap_handler as usize,
        );
        let pid = task_control_block.pid.0 as isize;
        add_task(task_control_block);
        pid
    } else {
```

我们在 process.rs 中加入以上片段来实现 spawn。使用"make run TEST=1"来进行样例测试，得到结果：



可以看到通过了样例测试。

实验练习 2:

现在我们要为我们的 os 实现一种带优先级的调度算法：stride 调度算法。

算法描述如下:

1. 为每个进程设置一个当前 stride，表示该进程当前已经运行的“长度”。另外设置其对应的 pass 值（只与进程的优先权有关系），表示对应进程在调度后，stride 需要进行的累加值。
2. 每次需要调度时，从当前 runnable 态的进程中选择 stride 最小的进程调度。对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass。
3. 一个时间片后，回到上一步骤，重新调度当前 stride 最小的进程。

由于 priority ⩾2， 因而 pass ⩽ BIG_STRIDE/2， 在不考虑溢出的情况下， STRIDE_MAX – STRIDE_MIN ⩽ BigStride / 2， 始终成立， 这一点用反证法可以证明。 但现实中必然需要考虑溢出， 那么这里可以利用符号整数的特性， 避免在溢出后某些本应延后调度的进程因为 stride 值溢出后从小值开始的缘故被优先调度。 所以， 可以将 BIG_STRIDE 设置为 u64 类型最大值，结合 STRIDE_MAX – STRIDE_MIN ⩽ BigStride / 2 亦会有 STRIDE_MIN – STRIDE_STRIDE >= -BigStride / 2， 这个与 i64 的表示方式非常相似， 因而可以这样设计 stride 的比较方式:

我自己尝试了去实现 BinaryHeap 进行调度， 但是程序会卡死在 initproc 中， 作为一个 Rust 新手以及确实很久没接触算法了， 不知道什么地方除了问题， 于是就用了最简单的思路，在每次 fetch 下一个 task 之前对这个队列中的 task 做检查，找到最小的 stride。这里每次将初始最小值设定为 0x7FFF_FFFF 是为了解决前述的溢出问题， 本质上利用有符号整数的思想就是把大于 MAX/2 等价为负数。 这样 TIPs 中说明的 使用 8 bits 存储 stride, BIG_STRIDE = 255，则: (125 < 255) == false, (129 < 255) == true 就好理解了。

```rust
/// A simple FIFO scheduler.
impl TaskManager {
    /// Create 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
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        let mut min_index = 0;
        let mut min_stride = 0x7FFF_FFFF;
        for (idx, task) in self.ready_queue.iter().enumerate() {
            let inner = task.inner.exclusive_access();
            if inner.get_status() == TaskStatus::Ready {
                if inner.stride < min_stride {
                    min_stride = inner.stride;
                    min_index = idx;
                }
            }
        }

        if let Some(task) = self.ready_queue.get(min_index) {
            let mut inner = task.inner.exclusive_access();
            inner.stride += BIG_STRIDE / inner.priority;
        }
        self.ready_queue.remove(min_index)
    }
}

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

在 manager.rs 中加入这段代码，再在其他部分将相应的部分做修改，就可以实现该共功能。