

操作系统课程实验报告

rcore 第五章

陈林峰 1120192739

2021 年 12 月 24 日

目录

1 实验环境

1. OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
2. Kernel: x86_64 Linux 5.10.16.3-microsoft-standard-WSL2
3. Rust: rustc 1.58.0-nightly (936f2600b 2021-11-22)

2 实验目的

已完成的操作系统内核已经提供了很多功能，但其还存在许多缺陷，其一就是其不能动态地加载到内存中，我们将所有的代码直接嵌入到内存中，而且无法对应用程序的执行进行删改，而且现代操作系统中非常重要的进程概念还没有实现，这一章就需要实现进程这个抽象，而为了完成这个抽象，需要实现三个重要的系统调用，包括 fork、waitpid、exec，这三个系统调用提供了创建进程，等待进程和执行进程等功能。同时，为了实现动态加载应用程序到内存中的功能，我们需要实现一个简易的 Shell 程序，而要实现这个终端程序，就需要完成 read 这个从键盘获取输入的系统调用。

3 实验步骤

- 重新实现应用程序加载器，实现根据应用名加载应用程序
- 将原有的任务管理器进行分割
- 定义进程控制块，将原来的功能重新编排，并为其实现自动回收机制
- 完成 fork,waitpid,exec 三个系统调用
- 实现 read 系统调用

4 实验内容

4.1 进程

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。其是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本分配单元也是基本执行单元。

4.2 进程和线程关系

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度。

4.3 父进程与子进程

一个进程通过 fork 调用，内核会新建一个进程，新的进程拥有和父进程一样的东西，包括数据，寄存器内容，但这些东西是位于不同的地址空间中的。在返回用户态的那一瞬间，两个进程只有保存 fork 返回值的寄存器值不相同，对于原进程来说返回的是新的进程的 pid，对于新的进程来说，其返回值为 0。将原进程称为父进程，新进程称为子进程。

当一个子进程通过系统调用 exit 退出时，操作系统不能立即回收其所有资源，因为其还要继续系统调用处理，如果回收其资源例如内核栈的化，就会导致处理不能继续。因此可以在进程退出时只回收一部分不再使用的资源，然后由其父进程通过 waitpid 系统调用来收集子进程的退出状态然后回收剩余的资源。

一个进程可以通过 exec 系统调用来执行其它可执行文件，这个系统调用负责的工作就是将当前进程地址空间的数据清除，将新的可执行文件加载到地址空间中。

4.4 初始进程与 Shell 进程

初始进程负责创建子进程 User_shell, 并不断地进行子进程的回收工作。User_Shell 进程负责从控制台读取输入的字符，当其构成一个字符串时，就会尝试 fork 一个子进程并运行这个可执行程序。

User_Shell 的实现如下

```
fn main() -> isize {
    println!("The User Shell");
    let mut process_name = String::new();
    print!("GodOS#");
    loop {
        let ch = getchar();
        match ch {
            LF | CR => {
                //回车或换行时
                println!("");//换行
                if !process_name.is_empty() {
                    process_name.push('\0');
                    let pid = fork();
                    if pid == 0 {
                        //子进程
```

```

        let info = exec(process_name.as_str());
        if info == -1 {
            //执行失败
            println!("The error occurs when executing");
            return -4;
        }
    } else {
        //父进程
        let mut exit_code: i32 = 0;
        //等待子进程完成
        let exit_pid = wait_pid(pid as usize, &mut exit_code);
        if exit_pid == pid {
            println!("Shell: Process {} exited with code {}", pid, exit_code);
        }
    }
    process_name.clear();
}
print!("GodOS#");
}
DEL | BS => {
    //退格键
    if !process_name.is_empty() {
        process_name.pop(); //删除最后一个字符
        print!("{}", BS as char); //移动光标往前一个字符
        print!(" ");
        print!("{}", BS as char);
    }
}

_ => {
    process_name.push(ch as char);
    print!("{}", ch as char); //打印在屏幕上
}
}
}
}
}

```

4.5 基于应用名的加载器

在之前的实现中，我们使用的是基于应用编号的加载器，根据应用名称的加载器只需要稍加修改，第一步就是在 link-app.S 的生成代码中增加一部分内容用来存放各个应用的名称，第二步在原来基于应用编号获取应用程序数据的基础上修改为查找名称所在的编号即可。

4.6 进程控制块

在完成进程控制块的修改之前，需要将进程标识符抽象出来，与前面的物理页帧分配器一样，我们也为其实现 alloc,dealloc 并为其实现自动回收机制。其抽象也很简单只是包装了 usize 基本数据类型。接下来就是完

成内核栈空间的分配，之前我们根据应用的编号从内核的高地址部分为应用程序分配栈空间，现在需要根据其进程标识符来分配空间。同时也为其实现了自动回收机制，这需要在之前定义的 MemorySet 地址空间的定义新的功能，即将内核栈对应的物理页帧回收掉，其实现如下：

```
pub fn remove_from_startaddr(&mut self, startaddr: VirtAddr) {
    //从一个起始地址找到对应的段，将这个段对应的页删除
    let virtpage: VirtPageNum = startaddr.into();//转换为虚拟页号
    if let Some((index, area)) = self
        .areas
        .iter_mut()
        .enumerate()//根据每一个内存区域的起始页号找到对应的area
        .find(|(_index, maparea)| maparea.vpn_range.get_start() == virtpage)
    {
        area.unmap(&mut self.page_table);//解除原来的映射
        self.areas.remove(index);//从area中将其删除
    }
}
```

进程控制块抽象如下：

```
pub struct TaskControlBlock {
    //不可变数据
    pub pid: PidHandle,
    pub kernel_stack: KernelStack,
    //可变数据
    inner: MyRefCell<TaskControlBlockInner>,
}
pub struct TaskControlBlockInner {
    pub task_cx_ptr: TaskContext,           //任务上下文栈顶地址的位置,位于内核空间中
    pub task_status: TaskStatus,
    pub memory_set: MemorySet,             //任务地址空间
    pub trap_cx_ppn: PhysPageNum,         //trap上下文所在的物理块
    pub base_size: usize,                 //应用程序的大小
    pub exit_code: isize,                 //保存退出码
    pub parent: Option<Weak<TaskControlBlock>>, //父进程
    pub children: Vec<Arc<TaskControlBlock>>, //子进程需要引用计数

    pub stride: usize, //已走步长
    pub pass: usize, //每一步的步长,只与特权级相关
}
```

进程控制块的创建与之前的大致相同，只是现在的内核栈是根据 pid 来申请。

4.7 任务管理器

任务管理器功能被划分为 TaskManager 与 Processor, 前者负责管理不在运行状态的进程，提供加入与弹出的功能，即可以向其进程队列添加一个进程，也可以弹出一个进程放入 Processor 队列中。其实现如下：

```
pub struct TaskManager {
```

```

//进程管理器，负责管理所有的进程
//使用双端队列和引用计数进行管理，如果不将任务控制块移到
//堆上进行存储，任务管理器只保留指针，那么在移动任务控制块时会
//带来性能损耗
task_ready_queue: VecDeque<Arc<TaskControlBlock>>,
}

pub fn add(&mut self, task: Arc<TaskControlBlock>) {
    //添加一个进程控制块
    self.task_ready_queue.push_back(task)
}

pub fn pop(&mut self) -> Option<Arc<TaskControlBlock>> {
    self.task_ready_queue.pop_front() //FIFO, 先进先出调度
}

```

后者负责管理当前处理机上运行的进程，并维护了一个全局的内核任务上下文用来进行任务切换，其提供的功能包括将当前进程从处理机移出并从 TaskManager 获取一个来执行，拷贝一份当前进程，其具体实现如下：

```

pub struct Processor {
    //当前cpu执行的进程
    current: Option<Arc<TaskControlBlock>>,
    //进程切换上下文
    //这是一个特殊的进程切换上下文，用于从当前的任务管理器中选择一个任务进行执行
    idle_task_cx_ptr: TaskContext,
}

///idle控制流的作用是将进程切换隔离开来，这样换入换出进程时所用的栈是不一样的
/// idle控制流用于进程调度，其位于内核进程的栈上，而换入换出是在应用的内核栈进行
pub fn run() {
    //在内核初始化完成之后需要开始运行
    loop {
        let mut processor = PROCESSOR.lock();
        if let Some(task) = fetch_task() {
            //从任务管理器成功弹出一个任务
            let mut task_inner = task.get_inner_access();
            let next_task_cx_ptr = &task_inner.task_cx_ptr as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;

            // INFO!("[kernel] find the nex task PID:{},task.get_pid());
            drop(task_inner); //释放掉获取的引用，因为要切换进程了
            processor.current = Some(task);
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            drop(processor); //释放引用
            unsafe {
                __switch(idle_task_cx_ptr, next_task_cx_ptr);
            }
        }
    }
}

```

```
pub fn schedule(last_task_cx_ptr: *mut TaskContext) {
    //当时间片用完或者是进程自行放弃cpu时，需要切换进程
    // 这时需要切换到内核进行进程切换的idle控制流上，在
    //上面的began_run_task中，当内核开始运行第一个进程时，
    //就会在内核栈上形成自己的任务上下文，其返回时继续进行
    //循环查找下一个进程
    let processor = PROCESSOR.lock();
    let idle_task_cx_ptr = &processor.idle_task_cx_ptr as *const TaskContext;
    drop(processor);
    // DEBUG!("[kernel] schedule");
    unsafe {
        __switch(last_task_cx_ptr, idle_task_cx_ptr);
    }
}
```

4.8 fork 系统调用

要完成 fork 系统调用，首先需要完成对应用地址空间中的各个 MapArea 的拷贝，即需要拷贝虚拟页的范围，控制权限，映射方式等。在得到了这个拷贝之后，就需要将对应的数据拷贝过来，这就需要遍历应用程序地址空间中的所有页，并将其上面的所有东西都相应拷贝到新的进程地址空间中。其实现如下：

```
pub fn from_existed_memset(src_memset: &MemorySet) ->
Self {
    //从一个已经存在的地址空间拷贝一份
    let mut memoryset = MemorySet::new_bare();
    memoryset.map_trampoline(); //映射跳板页，跳板页并没有加入到地址空间中，需要单独映射
    for area in src_memset.areas.iter() {
        let new_area = MapArea::copy_from_other(area); //拷贝一个maparea
        memoryset.push(new_area, None); //
        for vpn in area.vpn_range {
            let src_data = src_memset.translate(vpn).unwrap().ppn(); //获取父进程的虚拟页的对应的物理页
            let dis_data = memoryset.translate(vpn).unwrap().ppn(); //获取子进程虚拟页对应的物理页
            dis_data
                .get_bytes_array() //获取字节数组
                .copy_from_slice(src_data.get_bytes_array()); //拷贝数据
        }
    }
    memoryset
}
```

fork 的实现有点类似于之前的 TaskControlBlock 的 new 函数，不同之处在于子进程的地址空间是由上面的函数创建而来，而不是解析 ELF 数据，而且需要在在其父进程中插入，并修改其内核栈的位置。

```
parent_inner.children.push(task_control_block.clone());
//将子进程放入父进程的孩子节点中
let trap_cx = task_control_block.get_inner_access().get_trap_cx();
//获取子进程的TrapFrame并修改其内核栈顶的位置
trap_cx.kernel_sp = kernel_stack_top;
```

为了实现一个调用，两个返回，我们需要在 `sys_fork()` 这个系统调用处理函数动手脚，简单来说，就是将创建的子进程的 `TrapFrame` 的 `a0` 寄存器给修改为 0，而对于父进程来说，我们在完成子进程创建之后，再将其 `a0` 寄存器的值修改为子进程的 `pid` 号。

```
pub fn sys_fork()->isize{
    .....
    let trap_cx_ptr= new_task.get_inner_access().get_trap_cx();
    trap_cx_ptr.reg[10] = 0; //对于子进程来说，其返回值为0
    add_task(new_task);
    new_pid as isize //对于父进程来说，其返回值为子进程的pid
}
```

4.9 exec 系统调用

`exec` 的实现比较简单，我们只需要根据新的 ELF 文件生成一个新的地址空间，再将当前的地址空间更换掉，并重新写入 `TrapFrame` 内容，包括新的程序入口，用户栈顶位置，内核 `satp` 以及内核栈顶位置。这样在这个系统调用返回后就会加入新的应用程序入口，开启新的执行了。

4.10 read 系统调用

```
pub fn sys_read(fd:usize,buf:*const u8,len:usize)->isize{
    match fd {
        FD_STDIN=>{
            let mut c :usize;
            loop {
                c = console_getchar();
                if c==0 {
                    suspend_current_run_next(); //阻塞切换任务
                    continue;
                }
                else {
                    break
                }
            }
            let ch = c as u8;
            let mut buffer = translated_byte_buffer(current_user_token(),buf,len);
            unsafe {
                //写入用户地址空间的缓冲区中
                buffer[0].as_mut_ptr().write_volatile(ch);
            }
            1 //返回用户空间
        }
        _ =>{
            panic!("Unsupport fd");
        }
    }
}
```

```
}
}
```

从上面的代码可以看到, `sys_read` 的内核实现是阻塞的, 如果是输入换行符或者没有输入, 那么 `sys_read` 将会一直处于内核态等待着输入, 只有输入字符才会返回到用户态进行判断。

4.11 waitpid 系统调用

```
pub fn exit_current_run_next(exit_code: isize){
    //终止当前任务运行下一个任务
    //获得当前cpu执行的任务
    let current_task = take_current_task().unwrap();
    let mut current_task_inner = current_task.get_inner_access();
    //标记僵尸进程
    current_task_inner.task_status = TaskStatus::Zombie;
    //保存返回码
    current_task_inner.exit_code = exit_code;
    {
        let mut init_proc_inner = INITPROC.get_inner_access();
        for child in current_task_inner.children.iter(){
            //挂载到初始进程的孩子上
            child.get_inner_access().parent = Some(Arc::downgrade(child));
            init_proc_inner.children.push(child.clone());
        }
    }
    //自动解除引用
    current_task_inner.children.clear(); //释放子进程的引用计数
    current_task_inner.memory_set.clear_area_data(); //提前释放掉地址空间
    drop(current_task_inner);
    drop(current_task); //去掉当前进程的引用, 相当于销毁了进程
    let mut _unused = TaskContext::zero_init();
    schedule(&mut _unused as *mut TaskContext); //重新调度
}
```

在当前进程执行退出或发生错误退出时, 我们并没有将其所有的资源回收, 而只是将其标记为一个僵尸进程, 并将其所有的孩子进程挂载到初始进程 `INITPROC` 上面, 并删除其自身对所有孩子进程的引用, 通过提前删除进程所拥有的 `MapArea`, 我们可以提前获取更多可用资源, 因为这一部分的内容不会再被使用。

而剩余的资源, 则需要子进程的父进程或者是初始进程来对其进行完全回收, 这就需要 `waitpid` 系统调用

```
pub fn sys_waitpid(pid: isize, exit_code_ptr: *mut i32) -> isize{
    let current_task = copy_current_task().unwrap();
    //获取正在执行的进程
    let mut task_inner = current_task.get_inner_access();
    if task_inner.children.iter()
        .find(|task| pid == -1 || pid as usize == task.get_pid())
        .is_none(){
        return -1;
    } //查找是否有对应的子进程或者是pid == -1
```



```

let pair = task_inner.children.iter()
    .enumerate()
    .find(|(_index, val)|{
        val.get_inner_access().is_zombie() && (pid == -1 || pid as usize == val.get_pid())
    });
if let Some((idx, _)) = pair{
    // 移除子进程
    let child = task_inner.children.remove(idx);
    assert_eq!(Arc::strong_count(&child), 1); // 确保此时子进程的引用计数为1
    let found_pid = child.get_pid(); // 子进程的pid
    let exit_code = child.get_inner_access().exit_code; //
    // 向当前执行的进程的保存返回值位置写入子进程的返回值
    *translated_refmut(task_inner.memory_set.token(), exit_code_ptr) = exit_code as i32;
    found_pid as isize // 返回找到的子进程pid
}
else { -2 }
}

```

通过查找子进程中的 pid 是否有与传入的 pid 相同或者确认传入的 pid 为-1 的情况，排除掉传入 pid 错误的情况，再进一步查看子进程是否有僵尸进程，如果有僵尸进程，则获取其退出值并写入当前进程的缓冲区，如果没有僵尸进程，就返回-2 告诉父进程没有僵尸进程的存在。

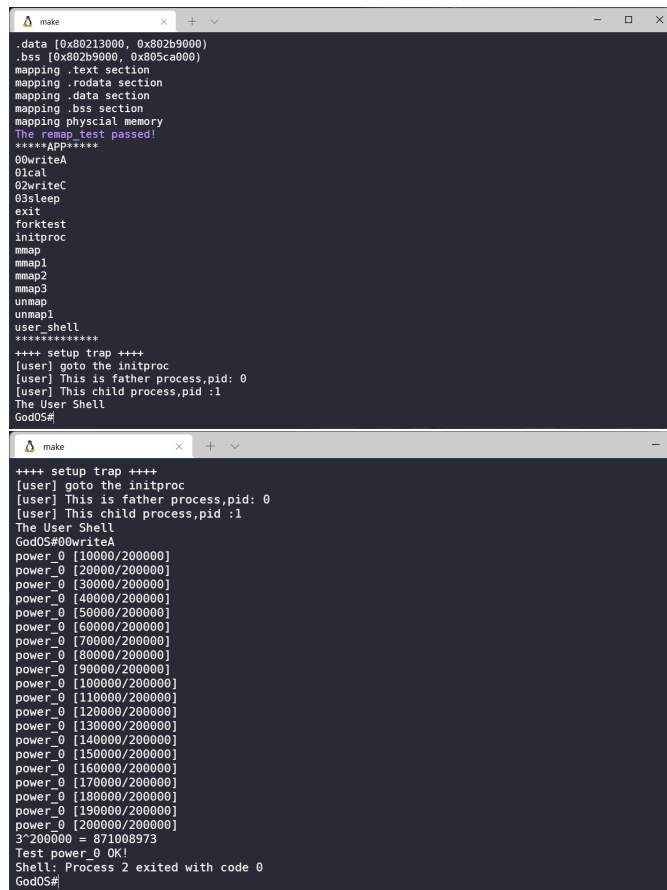
5 实验结果

如图所示，在完成初始化工作后，系统会将各个应用名称显示出来，我们可以在下方的输入行中输入想要运行的应用。这里我们选择应用 00writeA, 得到的结果如图所示1。

6 其它工作

实现了 fork 与 exec 系统调用后，理论上来说我们已经完成了一个进程想创建一个子进程的功能，但由于需要两次系统调用显得麻烦，于是我们可以实现一个 spawn 系统调用，其可以直接创建一个子进程并允许，即将 fork 与 exec 的功能结合起来，但这里与之前不同之处在于不再需要复制父进程的地址空间，而是直接根据传入的应用名称找到对应的可执行文件建立地址空间。

本次实验还对整个系统的结构做了总结，并展示了主程序的一个运行过程和 fork 系统调用的过程。



The image displays two terminal windows, both titled 'make'. The top window shows the initial execution of the program, including memory mapping for .data, .bss, .text, .rodata, and .data sections, and physical memory mapping. It then prints 'The remap_test passed!' and enters a loop of operations: 00writeA, 01cal, 02writeC, 03sleep, exit, forktest, initproc, mmap, mmap1, mmap2, mmap3, unmap, unmap1, and user_shell. The bottom window shows the continuation of the program after a trap is set. It prints 'The User Shell' and a series of memory addresses for power_0, ranging from 10000 to 200000. It then prints '3^200000 = 871008973', 'Test power_0 OK!', and 'Shell: Process 2 exited with code 0'.

```
.data [0x80213000, 0x802b9000)
.bss [0x802b9000, 0x805ca000)
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physcial memory
The remap_test passed!
****App****
00writeA
01cal
02writeC
03sleep
exit
forktest
initproc
mmap
mmap1
mmap2
mmap3
unmap
unmap1
user_shell
*****
**** setup trap ****
[user] goto the initproc
[user] This is father process,pid: 0
[user] This child process,pid :1
The User Shell
GodOS#

**** setup trap ****
[user] goto the initproc
[user] This is father process,pid: 0
[user] This child process,pid :1
The User Shell
GodOS#00writeA
power_0 [10000/200000]
power_0 [20000/200000]
power_0 [30000/200000]
power_0 [40000/200000]
power_0 [50000/200000]
power_0 [60000/200000]
power_0 [70000/200000]
power_0 [80000/200000]
power_0 [90000/200000]
power_0 [100000/200000]
power_0 [110000/200000]
power_0 [120000/200000]
power_0 [130000/200000]
power_0 [140000/200000]
power_0 [150000/200000]
power_0 [160000/200000]
power_0 [170000/200000]
power_0 [180000/200000]
power_0 [190000/200000]
power_0 [200000/200000]
3^200000 = 871008973
Test power_0 OK!
Shell: Process 2 exited with code 0
GodOS#
```

图 1: 应用名称显示与输入

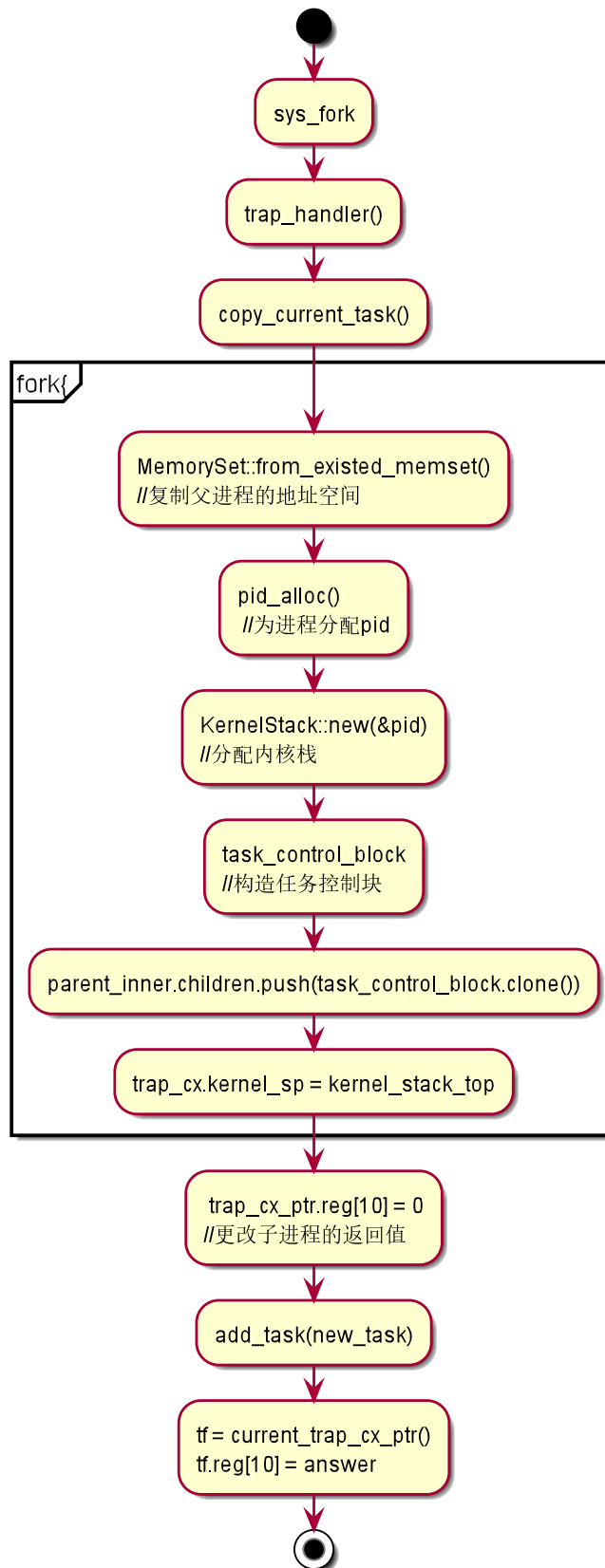


图 2: fork 过程

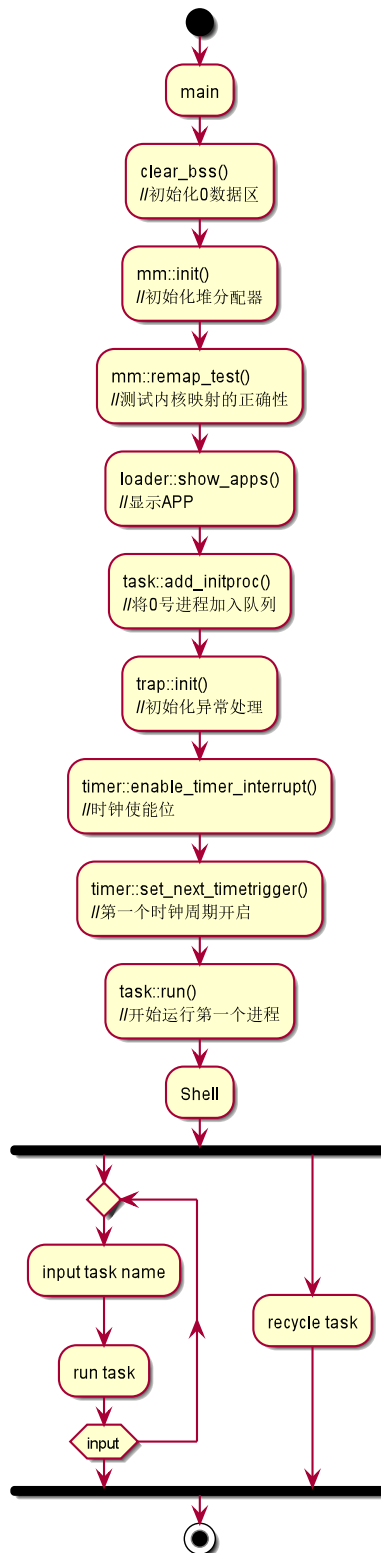


图 3: 系统运行过程

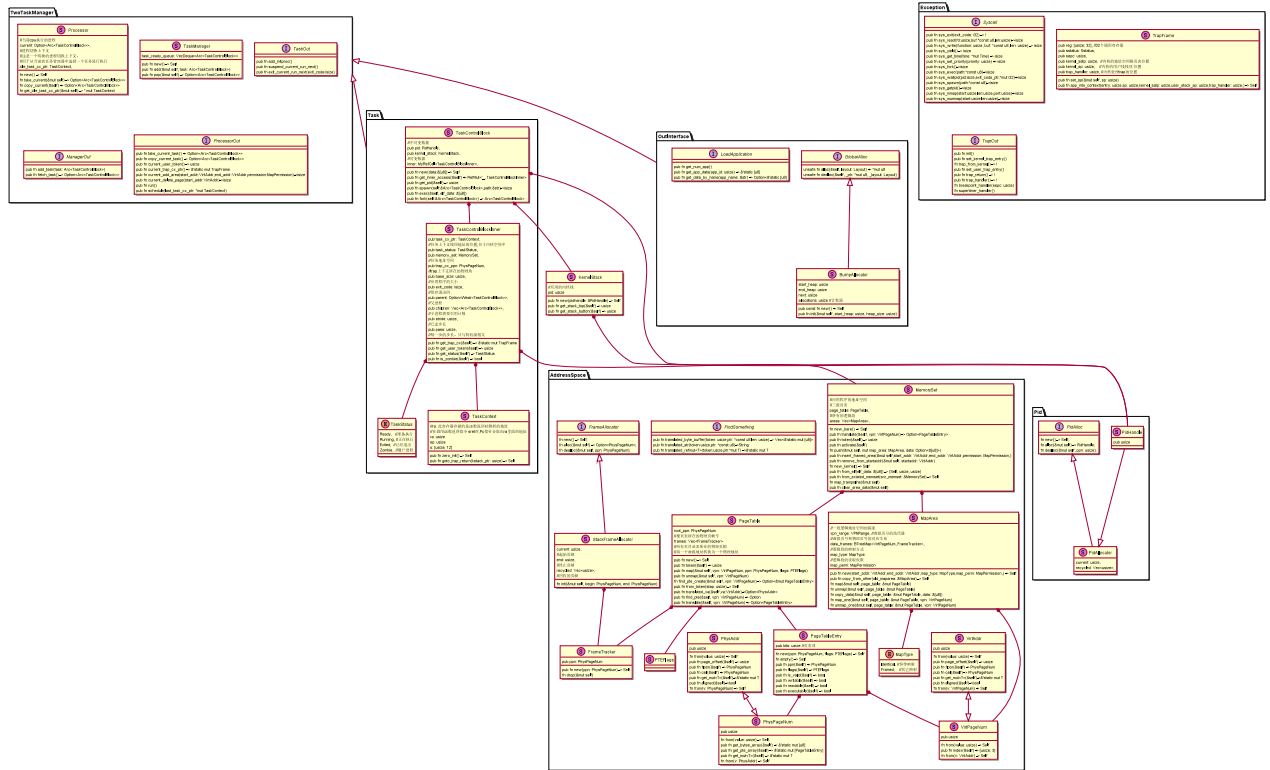


图 4: 系统结构