



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验四：用户程序与系统调用

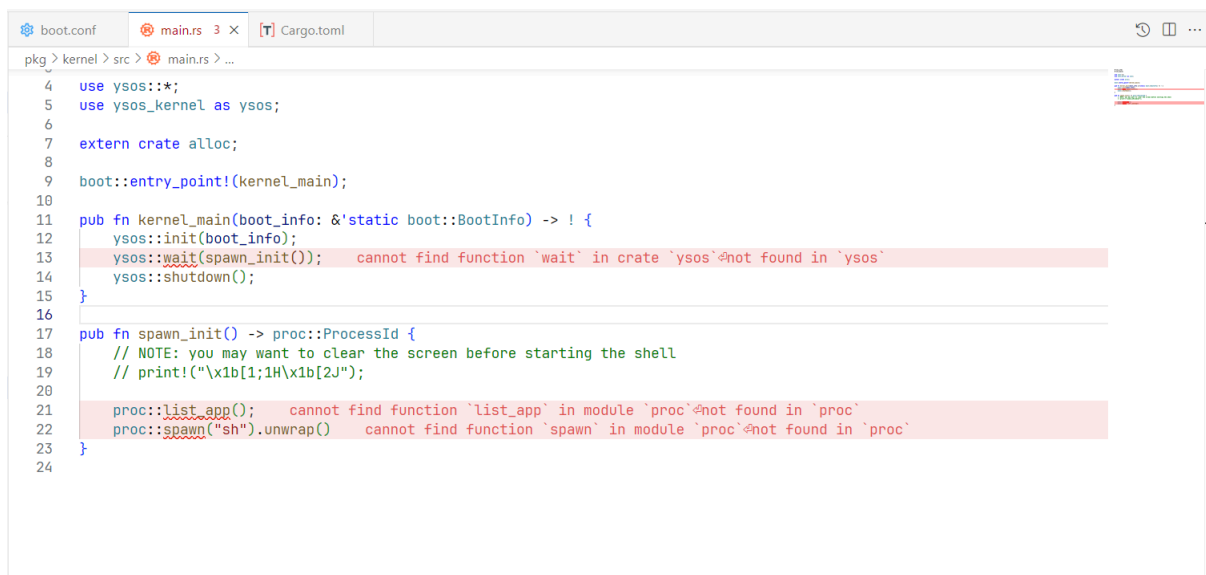
姓 名： 刘家祥
学 号： 23336152
教学班号： 计科二班
专 业： 计算机科学与技术
院 系： 计算机学院

2024~2025 学年第二学期

用户程序与系统调用

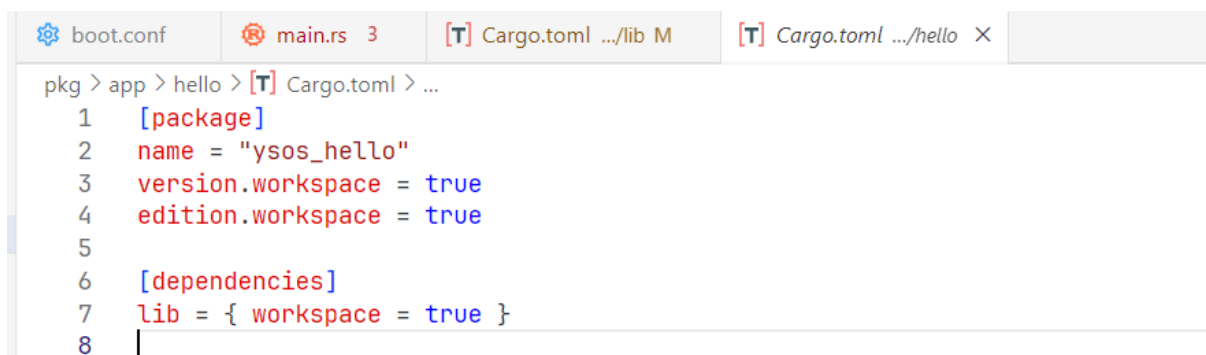
一. 用户程序

1. 编译用户程序



```
boot.conf  main.rs 3 x  Cargo.toml
pkg > kernel > src > main.rs > ...
4 use ysos::*;
5 use ysos_kernel as ysos;
6
7 extern crate alloc;
8
9 boot::entry_point!(kernel_main);
10
11 pub fn kernel_main(boot_info: &'static boot::BootInfo) -> ! {
12     ysos::init(boot_info);
13     ysos::wait(spawn_init()); cannot find function `wait` in crate `ysos` not found in `ysos`
14     ysos::shutdown();
15 }
16
17 pub fn spawn_init() -> proc::ProcessId {
18     // NOTE: you may want to clear the screen before starting the shell
19     // print!("{}", "\x1b[1;1H\x1b[2J");
20
21     proc::list_app(); cannot find function `list_app` in module `proc` not found in `proc`
22     proc::spawn("sh").unwrap() cannot find function `spawn` in module `proc` not found in `proc`
23 }
24
```

经过查看后续文档确认，这部分报错的函数除了 `wait` 都是等待实现的，所以这些错误暂时先不管。首先我们确认一下用户程序正确引用了用户态库：



```
boot.conf  main.rs 3  Cargo.toml .../lib M  Cargo.toml .../hello x
pkg > app > hello > Cargo.toml > ...
1 [package]
2 name = "ysos_hello"
3 version.workspace = true
4 edition.workspace = true
5
6 [dependencies]
7 lib = { workspace = true }
8
```

可以看到已经正确引用：

```
[dependencies]
lib = { workspace = true }
```

接下来我们尝试在 `pkg/app/hello` 目录下执行 `cargo build --release` 指令编译用户程序

hello :

```
camellia@LAPTOP-Camellia:~/ysos/0x04/pkg/app/hello$ cargo build --release
Compiling proc-macro2 v1.0.94
Compiling unicode-ident v1.0.18
Compiling core v0.0.0 (/home/camellia/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core)
Compiling compiler_builtins v0.1.151
Compiling quote v1.0.40
Compiling syn v2.0.100
Compiling num_enum_derive v0.7.3
Compiling rustc-std-workspace-core v1.99.0 (/home/camellia/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/rustc-std-workspace-core)
Compiling alloc v0.0.0 (/home/camellia/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/alloc)
Compiling num_enum v0.7.3
Compiling ysos_syscall v0.4.0 (/home/camellia/ysos/0x04/pkg/syscall)
Compiling yslib v0.4.0 (/home/camellia/ysos/0x04/pkg/lib)
warning: unused variable: `pid`
--> pkg/lib/src/syscall.rs:34:21
34 | pub fn sys_wait_pid(pid: u16) -> isize {
    |                ^^^^^ help: if this is intentional, prefix it with an underscore: `_pid`
    = note: `[warn(unused_variables)]` on by default

warning: `yslib` (lib) generated 1 warning
Compiling ysos_hello v0.4.0 (/home/camellia/ysos/0x04/pkg/app/hello)
Finished `release` profile [optimized] target(s) in 14.74s
camellia@LAPTOP-Camellia:~/ysos/0x04/pkg/app/hello$
```

可以看到成功编译成功。

”在成功编译了用户程序后，用户程序将被脚本移动到 `esp/APP` 目录下，并以文件夹命名。辅助脚本 `ysos.py` 功能已完备，不过 `Makefile` 需要根据此次实验进行更新。“由于我们之前是用 `cargo build` 进行编译的，而现有的几个错误在不注释代码的情况下显然无法顺利使用 `python3 ysos.py run` 进行完整的编译，所以暂时先忽略这部分内容，等 `main.rs` 那几个函数实现了再看。

2. 加载用户程序

首先按照实验指导在 `pkg/kernel/config/boot.conf` 中，将 `load_apps` 设置为 `true`。

在 `pkg\boot\src\lib.rs` 中，定义一个 `App` 结构体，并添加“已加载的用户程序”字段到 `BootInfo` 结构体中：根据 `更好的类型声明？` 提示，我们选择：

- 使用 `const` 指定用户程序数组的最大长度。
- 尝试定义 `AppListRef` 类型，用于存储 `loaded_apps.as_ref()` 的返回值类型，可以只关心 `'static` 生命周期。
- 抛弃 `App` 类型的生命周期，直接声明 `ElfFile<'static>`。

不过我们在定义 `AppListRef` 类型的时候选择使用 `pub type AppListRef<'a> = Option<&'a AppList>;` 即用 `'a` 而不是 `'static`。当前的 `AppListRef<'a>` 定义使用了泛型生命周期 `'a`，这比只限定 `'static` 更灵活，并且也能覆盖 `'static` 的情况。

```

/// The maximum number of apps that can be loaded.
pub const MAX_APP_NUM: usize = 16;

/// App information
#[derive(Debug)]
pub struct App {
    /// The name of app
    pub name: ArrayString<16>,
    /// The ELF file
    pub elf: ElfFile<'static>,
}

/// A list of loaded apps.
pub type AppList = ArrayVec<App, MAX_APP_NUM>;

/// A reference to the list of loaded apps.
pub type AppListRef<'a> = Option<&'a AppList>;

```

- 在 `pkg/boot/src/fs.rs` 中，创建函数 `load_apps` 用于加载用户程序，并参考 `fs.rs` 中的其他函数，处理文件系统相关逻辑，补全代码，这里除了补全之外，我还对于变量命名、文件访问方式方式、目录和文件检查以及错误处理方面进行了一些改进，来提高整体代码的质量与可读性。（代码太长在最后给出）
- 接下来按照实验指导在 `boot/src/main.rs` 中，`main` 函数中加载好内核的 `ElfFile` 之后，根据配置选项按需加载用户程序，并将其信息传递给内核。（代码已给出，略）
- 修改 `ProcessManager` 的定义与初始化逻辑，将 `AppList` 添加到 `ProcessManager` 中。
- 最后修改 `kernel/src/proc/mod.rs` 的 `init` 函数。

2.1. 阶段性成果

这里遇到了找不到 app 的问题，根据输出的日志我发现即使加入 `load_app=true` 也并没有让内核尝试加载用户程序，因为 `config.rs` 文件里面的这句 `"load_apps" => self.load_apps = r10 != 0`，实际上是识别的 0 或 1，将 true 改为 1 就行了，为了方便后续使用我把那里改进了一下，能兼容 t/f 和 0/1:

```

"load_apps" => {
    // 增强布尔值解析，同时支持数字和字符串表示
    self.load_apps = match value.to_lowercase().trim() {
        "true" | "yes" | "1" => true,
        _ => r10 != 0, // 保持原有的数字解析逻辑
    }
},

```

```
PROBLEMS 17 OUTPUT DEBUG CONSOLE TERMINAL PORTS 7 python3 - 0x04 + - - - -  
  
r15: 0x0000000000000000, r14: 0x0000000000000000, r13: 0x0000000000000000,  
r12: 0x0000000000000000, r11: 0x0000000000000000, r10: 0x0000000000000000,  
r9 : 0x0000000000000000, r8 : 0x0000000000000000, rdi: 0x0000000000000000,  
rsi: 0x0000000000000000, rdx: 0x0000000000000000, rcx: 0x0000000000000000,  
rbx: 0x0000000000000000, rax: 0x0000000000000000, rbp: 0x0000000000000000,  
,  
vm: Some(  
  ProcessVm {  
    stack: Stack {  
      top: 0xfffff01fffff000,  
      bot: 0xfffff01ffe00000,  
    },  
    memory_usage: "2 MiB",  
    page_table: PageTable {  
      addr: PhysFrame[4KiB](0x5c01000),  
      flags: Cr3Flags(  
        0x0,  
      ),  
    },  
  },  
,  
)  
}  
[INFO ] Process Manager Initialized.  
[INFO ] Interrupts Enabled.  
[INFO ] YatSenOS initialized.  
[+] App list: hello  
[ERROR] ERROR: panic!  
  
PanicInfo {  
  message: not yet implemented: proc::spawn is not implemented yet,  
  location: Location {  
    file: "pkg/kernel/src/main.rs",  
    line: 24,  
    col: 5,  
  },  
  can_unwind: true,  
  force_no_backtrace: false,  
}  
}
```

可以看到 `[+]App list: hello` , 成功调用 `list_app` 函数并成功加载。

3. 生成用户程序

- 在 `kernel/src/proc/mod.rs` 中, 添加 `spawn` 和 `elf_spawn` 函数, 将 `ELF` 文件从列表中取出, 并生成用户程序。(代码已给出, 加到文件末尾就行了, 这里略)
- 在 `ProcessManager` 中, 实现 `spawn` 函数。

```
pub fn spawn(  
    &self,  
    elf: &ElfFile,  
    name: String,  
    parent: Option<Weak<Process>>,  
)
```

```

        proc_data: Option<ProcessData>,
    ) → ProcessId {
        let kproc = self.get_proc(&KERNEL_PID).unwrap();
        let page_table = kproc.read().clone_page_table();
        let proc_vm = Some(ProcessVm::new(page_table));
        let proc = Process::new(name, parent, proc_vm, proc_data);

        // 获取进程ID
        let pid = proc.pid();

        let mut inner = proc.write();
        // 加载ELF文件到进程页表, 设置用户访问权限标志
        if let Some(vm) = &mut inner.vm {
            let mapper = &mut vm.page_table.mapper();
            let frame_allocator = &mut *get_frame_alloc_for_sure();

            // 使用load_elf函数加载ELF文件到进程的页表, 并启用用户访问权限
            if let Err(err) = elf::load_elf(elf, 0, mapper, frame_allocator,
true) {
                error!("Failed to load ELF: {:?}", err);
                return pid;
            }

            // 为进程分配新栈
            let stack_top = vm.init_proc_stack(pid);
            // 设置栈帧, 第一条指令是ELF的入口点

            inner.context.init_stack_frame(VirtAddr::new(elf.header.pt2.entry_point()),
            stack_top);
        }

        // 标记进程为就绪状态
        inner.status = ProgramStatus::Ready;
        drop(inner);

        trace!("New {:?}", &proc);

        // 将进程添加到进程映射中
        self.add_proc(pid, proc.clone());

        // 将进程加入就绪队列
        self.push_ready(pid);

        pid
    }
}

```

- 修改 `init_stack_frame` 函数以正确使用刚刚添加的用户段选择子

```
// 获取用户段选择子
let selector = crate::memory::gdt::get_user_selector();
self.value.stack_frame.code_segment = selector.user_code_selector;
self.value.stack_frame.stack_segment = selector.user_data_selector;
```

- 在 `ProcessInner` 和 `ProcessVm` 中实现 `load_elf` 函数，来处理代码段映射等内容。

```
//ProcessInner

pub fn load_elf(&mut self, elf: &ElfFile) → Result<(), MapToError<Size4KiB>> {
    // 确保进程虚拟内存存在
    if self.proc_vm.is_none() {
        return Err(MapToError::ParentEntryHugePage);
    }

    // 委托给ProcessVm的load_elf函数处理
    self.vm_mut().load_elf(elf)
}
```

`ProcessVm` 实现部分代码太长放到最后。

- 在 GDT 中为 Ring 3 的代码段和数据段添加对应的选择子，在初始化栈帧的时候将其传入。

```
lazy_static! {
    static ref GDT: (GlobalDescriptorTable, KernelSelectors) = {
        let mut gdt = GlobalDescriptorTable::new();
        // ...

        // 添加Ring 3的代码段和数据段选择子
        let user_code_selector = gdt.append(Descriptor::user_code_segment());
        let user_data_selector = gdt.append(Descriptor::user_data_segment());

        // ...
    };
}
```

- 通过合适的方式暴露出来，以供栈帧初始化时使用：

```
// 获取用户段选择子
let selector = crate::memory::gdt::get_user_selector();
self.value.stack_frame.code_segment = selector.user_code_selector;
self.value.stack_frame.stack_segment = selector.user_data_selector;
```

- 关于加分项 1:

为了让代码更加模块化、有更好的可读性，我把主要实现代码转移到了 `data.rs` 中：

```

pub struct ProcessData {
    // shared data
    pub(super) env: Arc<RwLock<BTreeMap<String, String>>>,
    // Memory usage tracking
    pub(super) code_bytes: u64, // Bytes used by code/data segments
    pub(super) stack_pages: u64, // Pages used by stack
    pub(super) total_pages: u64, // Total pages used (code + stack + others if
any)
}

impl Default for ProcessData {
    fn default() → Self {
        Self {
            env: Arc::new(RwLock::new(BTreeMap::new())),
            code_bytes: 0,
            stack_pages: 0,
            total_pages: 0,
        }
    }
}

impl ProcessData {
    pub fn new() → Self {
        Self::default()
    }

    // Updates memory usage based on loaded ELF segments and stack size
    pub(super) fn update_memory_usage(&mut self, code_bytes: u64, stack_pages:
u64) {
        self.code_bytes = code_bytes;
        self.stack_pages = stack_pages;
        // Calculate total pages (assuming PAGE_SIZE is 4KiB)
        let code_pages = (code_bytes + crate::memory::PAGE_SIZE - 1) /
crate::memory::PAGE_SIZE;
        self.total_pages = code_pages + stack_pages;
    }

    // Returns total memory usage in bytes
    pub fn memory_usage_bytes(&self) → u64 {
        self.total_pages * crate::memory::PAGE_SIZE
    }

    // Returns total memory usage in pages
    pub fn memory_usage_pages(&self) → u64 {
        self.total_pages
    }
}

```


二. 系统调用的实现

1. 在 `src/interrupt/syscall/mod.rs` 中，补全中断注册函数，并在合适的地方调用它：

- 在 `idt` 的初始化中，注册 `0x80` 号中断的处理函数为 `syscall_handler`。 `int 0x80` 并非 `Irq` 中断，直接使用 `consts::Interrupts::Syscall` 即可。
- 与时钟中断类似，或许为系统调用准备一个独立的栈是个好主意？
- 使用 `set_privilege_level` 设置 `DPL` 为 3，以使用户态程序可以触发系统调用。

```
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    idt[consts::Interrupts::Syscall as usize]
        .set_handler_addr(syscall_handler as u64) // 设置处理函数地址
        .set_stack_index(gdt::SYSCALL_IST_INDEX) // 设置独立的系统调用栈（假设 GDT 中定义了 SYSCALL_IST_INDEX）
        .set_privilege_level(x86_64::PrivilegeLevel::Ring3); // 设置 DPL 为 3，允许用户态调用
}
```

```
// mod.rs
pub mod syscall;
// ...

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        // ...
        syscall::register_idt(&mut idt); // 注册系统调用中断
        // ...
    };
}
```

2. 与时钟中断类似，或许为系统调用准备一个独立的栈是个好主意？

```
pub const SYSCALL_IST_INDEX: u16 = 1; // 为系统调用定义 IST 索引
```

三. 用户态库的实现

1. 动态内存分配

为了方便用户态程序使用动态内存分配，而不是基于 `brk` 等方式进行完全用户态的动态内存管理，选择使用系统调用的方式，将内存分配的任务委托给内核完成。

与内核堆类似，在 `src/memory/user.rs` 中，定义了用户态的堆。

与内核使用 `static` 在内核 `bss` 段声明内存空间不同，由于在页表映射时需添加 `USER_ACCESSIBLE` 标志位，用户态堆需要采用内核页面分配的能力完成。其次需要注意的是，为了调试和安全性考量，这部分内存还需要 `NO_EXECUTE` 标志位。

有关用户态堆初始化的过程需要补全部分代码，指定合适的页面范围，分配并映射相关的页表，你可以参考、使用并修改 `elf::map_range` 完成这里的初始化。

参考 `elf::map_range` 函数来映射用户态堆页面：

```
pub fn init_user_heap() → Result<(), MapToError<Size4KiB>> {
    // ...
    // 使用map_range来分配和映射用户态堆
    let range_start = Page::containing_address(VirtAddr::new(USER_HEAP_START as
u64));
    let range_end = range_start + USER_HEAP_PAGE as u64;

    // 设置页面标志：存在、可写、用户可访问，不可执行
    let flags = PageTableFlags::PRESENT |
                PageTableFlags::WRITABLE |
                PageTableFlags::USER_ACCESSIBLE |
                PageTableFlags::NO_EXECUTE;

    for page in Page::range(range_start, range_end) {
        let frame = frame_allocator
            .allocate_frame()
            .ok_or(MapToError::FrameAllocationFailed)?;
        unsafe {
            mapper
                .map_to(page, frame, flags, frame_allocator)?
                .flush();
        }
    }

    // ...
}
```

2. 标准输入输出

- 在 `src/proc/data.rs` 中, 修改 `ProcessData` 结构体, 类似于环境变量的定义, 添加一个“文件描述符表”:

```
pub(super) resources: Arc<RwLock<ResourceSet>>,
```

- 在 `ProcessData` 的 `default` 函数中初始化, 添加默认的资源:

```
resources: Arc::new(RwLock::new(ResourceSet::default()))
```

- 之后添加 `read` 和 `write` 函数, 用于在系统调用中根据文件描述符进行读写:

```
pub fn read(&self, fd: u8, buf: &mut [u8]) → isize {
    self.resources.read().read(fd, buf)
}

pub fn write(&self, fd: u8, buf: &[u8]) → isize {
    self.resources.read().write(fd, buf)
}
```

- 系统调用总是为当前进程提供服务, 因此可以在 `proc/mod.rs` 中对一些操作进行封装, 封装获取当前进程、上锁等操作:

```
pub fn read(fd: u8, buf: &mut [u8]) → isize {
    x86_64::instructions::interrupts::without_interrupts(||
get_process_manager().read(fd, buf))
}

pub fn write(fd: u8, buf: &[u8]) → isize {
    x86_64::instructions::interrupts::without_interrupts(||
get_process_manager().write(fd, buf))
}
```

3. 在 `interrupt/syscall/service.rs` 中, 你需要实现 `sys_write` 函数, 用于处理 `write` 系统调用, 使得用户程序得以进程输出:

- 使用 `core::slice::from_raw_parts` 将用户程序的缓冲区转换为 `&[u8]`。
- 将缓冲区传入资源的 `write` 方法中, 并返回写入的字节数。
- 在分发函数中使用 `context.set_rax` 设置返回值, 并调用 `sys_write` 函数。

```

pub fn sys_write(args: &SyscallArgs) → usize {
    let fd = args.arg0 as u8;
    let ptr = args.arg1 as *const u8;
    let len = args.arg2;
    if ptr.is_null() || len == 0 {
        return 0;
    }

    // 将指针和长度转换为切片
    let buf = unsafe { core::slice::from_raw_parts(ptr, len) };

    // 调用进程的write函数
    let result = write(fd, buf);

    // 如果结果为负数，返回0，否则返回写入的字节数
    if result.is_negative() {
        0
    } else {
        result as usize
    }
}

```

3.1. 实现 `sys_read` 函数

```

pub fn sys_read(args: &SyscallArgs) → usize {
    let fd = args.arg0 as u8;
    let ptr = args.arg1 as *mut u8;
    let len = args.arg2;
    if ptr.is_null() || len == 0 {
        return 0;
    }

    // 将指针和长度转换为可变切片
    let buf = unsafe { core::slice::from_raw_parts_mut(ptr, len) };

    // 调用进程的read函数
    let result = read(fd, buf);

    // 如果结果为负数，返回0，否则返回读取的字节数
    if result.is_negative() {
        0
    } else {
        result as usize
    }
}

```

4. 进程的退出

与内核线程防止再次被调度的“退出”不同，用户程序的正常结束，需要在用户程序中调用 `exit` 系统调用，以通知内核释放资源。

```
pub fn exit_process(args: &SyscallArgs, context: &mut ProcessContext) {  
    // 使用返回码退出进程  
    let ret_code = args.arg0 as isize;  
    exit(ret_code, context);  
}
```

通过补全 `exit_process` 函数实现了这个功能,获取退出码并调用 `exit` 函数来结束当前进程。

5. 进程的创建与等待

- 操作系统的内核入口点将能够简单实现如下功能：
 - 初始化内核
 - 生成 `init` 进程，并等待它退出
 - 关机
- 其中，等待进程退出的函数 `ysos::wait` 可以定义在 `kernel/src/lib.rs` 中：

```
pub fn wait(init: proc::ProcessId) {  
    loop {  
        if proc::still_alive(init) {  
            // Why? Check reflection question 5  
            x86_64::instructions::hlt();  
        } else {  
            break;  
        }  
    }  
}
```

并在 `kernel/src/proc/mod.rs` 中，补全 `still_alive` 函数(这里需要将 `ProcessManager` 里面的 `get_proc` 函数设置为 `pub`：

```
pub fn still_alive(pid: ProcessId) → bool {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // 检查进程是否仍然存活
        match get_process_manager().get_proc(&pid) {
            Some(proc) ⇒ {
                let status = proc.read().status();
                status ≠ ProgramStatus::Dead
            }
            None ⇒ false,
        }
    })
}
```

对于具体的进程操作、目录操作等功能，将会移步到用户态程序进行实现。为了给予用户态程序操作进程、等待进程退出的能力，这里还缺少最后几个系统调用需要实现：spawn、getpid 和 waitpid:

- spawn_process 函数实现了创建新进程的功能，它从系统调用参数中获取进程名称，然后尝试生成并运行该进程。

```
pub fn spawn_process(args: &SyscallArgs) → usize {
    // 获取应用名称
    let ptr = args.arg0 as *const u8;
    let len = args.arg1;

    if ptr.is_null() || len == 0 {
        return 0;
    }

    // 将输入参数转换为字符串
    let name = unsafe {
        let slice = core::slice::from_raw_parts(ptr, len);
        match core::str::from_utf8(slice) {
            Ok(s) ⇒ s,
            Err(_) ⇒ return 0,
        }
    };

    // 尝试生成进程
    match spawn(name) {
        Some(pid) ⇒ pid.0 as usize,
        None ⇒ 0,
    }
}
```

- `sys_waitpid` 函数实现了等待指定进程退出的功能,我使用了 `usize::MAX` 作为特殊值来表示进程仍在运行

```
pub fn sys_waitpid(args: &SyscallArgs) → usize {
    let pid = ProcessId(args.arg0 as u16);

    // 检查进程是否存活
    if !still_alive(pid) {
        // 如果进程已退出, 尝试获取退出码
        match get_exit_code(pid) {
            Some(code) ⇒ code as usize,
            None ⇒ 0, // 进程不存在或已被回收
        }
    } else {
        // 进程仍在运行, 返回特殊值表示正在运行
        usize::MAX // 使用最大的usize值表示进程仍在运行
    }
}
```

- 加入 `get_pid`, 这个函数是进程创建与等待部分的补充功能, 它允许用户程序获取自己的进程 ID。虽然在实验指导中没有详细描述具体实现, 但它是用户态程序操作进程所必需的基本系统调用之一。

```
pub fn sys_getpid(_args: &SyscallArgs) → usize {
    // 使用当前函数获取进程ID, 而不是直接使用processor模块
    let pid = crate::proc::get_process_manager().current().pid();
    pid.0 as usize
}
```

6. 阶段性成果

“至此为止, 你应当可以生成用户程序、输出 Hello, world!!!, 并正确退出。”如下图所示:

```
PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL PORTS 7 bash - 0x04 + - ... v x

[ INFO]: pkg/boot/src/main.rs@041: kernel_stack_size: 0x200,
[ INFO]: pkg/boot/src/main.rs@041: physical_memory_offset: 0xffff800000000000,
[ INFO]: pkg/boot/src/main.rs@041: kernel_path: "\\KERNEL.ELF",
[ INFO]: pkg/boot/src/main.rs@041: cmdline: "",
[ INFO]: pkg/boot/src/main.rs@041: load_apps: true,
[ INFO]: pkg/boot/src/main.rs@041: log_level: "info",
[ INFO]: pkg/boot/src/main.rs@041: }

  \  /  _  _  /  /  /  _  _  _  /  _  \  /  _  _  /
 \  /  _  _  \  /  \  \  _  _  \  _  \  /  _  _  \
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  \  /  \  /  \  /  \  /  \  /  \  /  \  /  \  /

                                     v0.4.0

[+] Serial Initialized.
[INFO ] Logger Initialized with level: info
[INFO ] Physical Offset   : 0xffff800000000000
[INFO ] Privilege Stack   : 0xffffffff000002b0d0-0xffffffff000002f0d0
[INFO ] Double Fault IST  : 0xffffffff000002f0d0-0xffffffff00000330d0
[INFO ] Page Fault IST    : 0xffffffff00000330d0-0xffffffff00000330d0
[INFO ] Timer IST         : 0xffffffff00000330d0-0xffffffff00000370d0
[INFO ] Syscall IST       : 0xffffffff00000370d0-0xffffffff00000370d0
[INFO ] Total IST size    : 64 KiB
[INFO ] Kernel Heap Size  : 8.000 MiB
[INFO ] Kernel Heap Initialized.
[INFO ] APIC initialized.
[INFO ] Interrupts Initialized.
[INFO ] Physical Memory    : 95.625 MiB
[INFO ] Free Usable Memory : 44.797 MiB
[INFO ] Frame Allocator initialized.
[INFO ] User Heap Initialized.
[INFO ] Process Manager Initialized.
[INFO ] Interrupts Enabled.
[INFO ] YatSenOS initialized.
[+] App list: hello
Hello, world!!!
[INFO ] YatSenOS shutting down.
camellia@LAPTOP-CameLLia:~/ysos/0x04$
```

四. 运行 shell

1. 编写自己的 Shell，作为用户与操作系统的交互方式，它需要实现一些必须功能：

列出当前系统中的所有用户程序 列出当前正在运行的全部进程 运行一个用户程序 同时，它也可以实现一些辅助的能力：

列出帮助信息 清空屏幕 … 为了实现一些信息的查看，你也需要实现如下两个系统调用：

`Syscall::Stat => {}`, `Syscall::ListApp => {}`,

shell 输入没能实现，暂时只是伪 shell：


```
al Help ← → 0x04 [WSL: Ubuntu-24.04]
PROBLEMS 28 OUTPUT DEBUG CONSOLE TERMINAL PORTS 7
bash - 0x04 + ▢ 🗑️ ... ▾ >

[INFO ] Total IST size   : 64 KiB
[INFO ] Kernel Heap Size : 8.000 MiB
[INFO ] Kernel Heap Initialized.
[INFO ] APIC initialized.
[INFO ] Interrupts Initialized.
[INFO ] Physical Memory   : 95.625 MiB
[INFO ] Free Usable Memory : 44.742 MiB
[INFO ] Frame Allocator initialized.
[INFO ] User Heap Initialized.
[INFO ] Process Manager Initialized.
[INFO ] Interrupts Enabled.
[INFO ] YatSenOS initialized.
[+] App list: hello, shell
欢迎使用YSOS Shell!
输入 help 获取帮助信息

[自动执行模式开始]
YSOS Shell - 可用命令:
  help      显示此帮助信息
  ls        列出所有可用的应用程序
  ps        列出当前运行的所有进程
  run <程序> 运行指定的程序
  clear     清空屏幕
  exit      退出Shell
学号: 23336152
可用的应用程序列表:
[+] App list: hello, shell
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Status |
  # 1 | # 0 | kernel       | 27    | 0      | 0 B | Ready
  # 2 | # 1 | shell        | 18    | 7      | 28 KiB | Running
Queue : [1]
CPUs  : [0: 2]
正在运行阶乘测试程序
错误: 无法运行阶乘测试程序
[自动执行模式结束]

ysos> [INFO ] YatSenOS shutting down.
camellia@LAPTOP-Camellia: ~/ysos/0x04$
```

然后发现有个 resource.rs 函数没有补全实现:

```

pub fn read(&mut self, buf: &mut [u8]) → Option<usize> {
    match self {
        Resource::Console(stdio) ⇒ match stdio {
            StdIO::Stdin ⇒ {
                // 缓冲区为空则尝试读取一个按键
                if buf.is_empty() {
                    return Some(0);
                }

                // 尝试读取一个按键
                if let Some(key) = input::try_pop_key() {
                    match key {
                        input::InputKey::Char(c) ⇒ {
                            let bytes = [c as u8];
                            buf[0] = bytes[0];
                            return Some(1);
                        }
                        input::InputKey::Backspace ⇒ {
                            // 对于退格键，返回退格的ASCII码(8)
                            buf[0] = 8;
                            return Some(1);
                        }
                        input::InputKey::Newline ⇒ {
                            // 对于换行键，返回回车的ASCII码(13)
                            buf[0] = b'\n';
                            return Some(1);
                        }
                    }
                }
            }
            _ ⇒ None,
        },
        Resource::Null ⇒ Some(0),
    }
}

```

补全之后发现就能正常运行并操作 shell 了，实现了如图所示的指令输出：

```
PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL PORTS 0
python3 - 0x04 + - - - - x

[INFO ] Process Manager Initialized.
[INFO ] Interrupts Enabled.
[INFO ] YatSenOS initialized.
ysos> help
Ysos Shell - 可用命令:
help      显示此帮助信息
ls        列出所有可用的应用程序
ps        列出当前运行的所有进程
run <程序> 运行指定的程序
clear     清空屏幕
exit     退出Shell
学号: 23336152
ysos> ps
当前运行的进程列表:
PID | PPID | Process Name | Ticks | Status |      |      |
# 1 | # 0 | kernel       | 41165 | 0      |      | 0 B | Ready
# 2 | # 1 | shell        | 41158 | 7      |      | 28 KiB | Running
Queue : [1]
CPUs   : [0: 2]
ysos> run hello
正在运行程序: hello
进程ID: 3
Hello, world!!!
程序 'hello' 已退出, 返回值: 233
ysos> ps
当前运行的进程列表:
PID | PPID | Process Name | Ticks | Status |      |      |
# 1 | # 0 | kernel       | 53668 | 0      |      | 0 B | Ready
# 2 | # 1 | shell        | 53661 | 7      |      | 28 KiB | Running
Queue : [1]
CPUs   : [0: 2]
ysos> 
```

五. 思考题

1. 是否可以在内核线程中使用系统调用？并借此来实现同样的进程退出能力？分析并尝试回答。

内核线程不能直接“使用”用户态意义上的系统调用，因为系统调用本质上是用户态程序请求内核服务的接口，涉及到特权级切换（从 Ring 3 到 Ring 0）。内核线程本身就运行在内核态（Ring 0，最高特权级），可以直接调用内核内部的函数来实现所需功能，无需经历特权级切换的开销和机制。

如果目标是实现与用户进程类似的“退出能力”，内核线程可以通过以下方式，而不是通过模拟系统调用：

1. **直接调用内核内部的清理函数**：当内核线程完成其任务后，它可以调用内核中专门用于资源回收和状态更新的函数。这比模拟系统调用更直接高效。
2. **状态标记与协作**：内核线程可以改变自身状态（例如，设置为“已完成”或“待清理”），并通知内核的其他部分（如调度器或特定的管理器）来完成后续的清理工作。
3. **不直接退出，而是转为休眠或等待状态**：许多内核线程是事件驱动的，完成一次任务后可能会进入休眠，等待新的事件唤醒，而不是完全“退出”。

因此，虽然内核线程不通过传统系统调用来实现退出，但内核自身有完善的机制来管理内核线程的生命周期和资源回收，可以达到类似的效果，但其实现方式和上下文与用户进程的系统调用完全不同。

2. 为什么需要克隆内核页表？在系统调用的内核态下使用的是哪一张页表？用户态程序尝试访问内核空间会被正确拦截吗？尝试验证你的实现是否正确。

2.1. 为什么需要克隆内核页表（更准确地说是为新进程创建独立的页表结构）？

1. **进程隔离**：每个进程都应有自己独立的地址空间。独立的页表是实现这种隔离的基础。即使内核空间的映射在所有进程中是共享的（指向相同的物理内存），每个进程也需要自己的页表结构来包含这些映射以及其私有的用户空间映射。当通过 `fork` 创建新进程时，通常会复制父进程的页表（或至少是用户空间的映射），或者采用写时复制（COW）策略。
2. **独立生命周期与管理**：进程的内存布局（如堆、栈）会动态变化。独立的页表允许每个进程独立管理其地址空间，而不会相互干扰。
3. **写时复制 (Copy-on-Write, COW)**：克隆页表是实现 COW 的前提。初始克隆时，父子进程可以共享物理页面，页表项标记为只读。当任一方尝试写入时，才会真正复制物理页面并更新各自的页表项，提高了 `fork` 的效率。

对于“克隆内核页表”，更精确的理解是：为每个新进程创建一个新的顶层页表（如 PML4），并将内核空间的标准映射复制到这个新页表中。用户空间的映射则根据创建方式（如 `fork` 或 `exec`）来设定。

2.2. 在系统调用的内核态下使用的是哪一张页表？

在系统调用期间，当 CPU 从用户态切换到内核态时，它仍然使用**当前发起系统调用的进程的页表**。这是因为内核代码需要能够访问该用户进程的地址空间，例如：

- 读取用户通过指针传递的参数。
- 将结果写回到用户空间的内存地址。

页表中通过权限位（如 x86 的 User/Supervisor 位）来区分用户态和内核态对页面的访问权限。内核态可以访问标记为 Supervisor 和 User 的页面，而用户态只能访问标记为 User 的页面。

2.3. 用户态程序尝试访问内核空间会被正确拦截吗？

是的，必须被正确拦截，这是操作系统内存保护的核心机制。页表项中的权限位（如 U/S 位）起着关键作用。如果一个页表项标记为 Supervisor-only（即内核态可访问，用户态不可访

问), 当用户态程序 (运行在 Ring 3) 尝试访问该页面时, MMU (内存管理单元) 会检测到权限冲突, 并产生一个页错误 (Page Fault) 异常 (通常是 #PF, 中断向量 0x0E)。操作系统的页错误处理程序会捕获这个异常。对于此类非法访问, 通常的处理方式是终止违规的用户进程, 以保护内核的完整性和系统的稳定性。

2.4. 尝试验证你的实现是否正确。

1. 验证页表克隆与隔离:

- 创建一个父进程, 然后 `fork` 一个子进程。
- 在子进程中修改一块用户内存区域的数据。
- 在父进程中检查同一虚拟地址的内存区域, 其数据不应受子进程修改的影响 (如果实现了 COW, 则在子进程写入时发生复制)。
- 验证子进程和父进程都能正常进行系统调用, 间接说明其内核空间映射是有效的。

2. 验证系统调用时使用的页表:

- 设计一个系统调用, 它接受一个用户空间指针作为参数, 并向该指针指向的内存写入一个值。
- 在用户程序中调用该系统调用, 并传入一个用户空间缓冲区的地址。
- 检查用户程序中该缓冲区的内容是否被内核正确修改。如果成功, 说明内核在系统调用期间正确使用了当前进程的页表来访问用户空间。

3. 验证用户态访问内核空间拦截:

- 编写一个用户程序, 尝试读取或写入一个已知的内核空间地址 (例如, 内核代码段的某个地址, 或者一个已知的内核数据结构地址, 确保该地址在页表中标记为 Supervisor-only)。
- 运行该用户程序。
- 预期的结果是: 该用户程序会因触发 Page Fault 而被操作系统终止。可以通过调试器观察异常, 或检查系统日志/内核打印的错误信息来确认。例如, 在 QEMU 中运行时, 可能会看到类似 “General Protection Fault” 或 “Page Fault” 的错误, 并指示出错的地址和原因。

3. 为什么在使用 `still_alive` 函数判断进程是否存活时, 需要关闭中断? 在不关闭中断的情况下, 会有什么问题?

在使用 `still_alive` 函数判断进程是否存活时, 需要关闭中断 (例如使用 `x86_64::instructions::interrupts::without_interrupts`) 主要是为了防止竞态条件 (Race Conditions) 和保证数据一致性。

3.1. 为什么需要关闭中断?

1. **原子性访问进程状态**: 进程的状态 (如 `Running` , `Ready` , `Blocked` , `Dead`) 存储在进程控制块 (PCB) 或相关数据结构中。这些数据结构是内核的共享资源。`still_alive` 函数需要读取这个状态。如果在读取过程中发生中断, 中断处理程序 (尤其是调度器或导致进程状态改变的其他内核服务) 可能会修改这个状态。关闭中断可以确保从读取状态到函数返回一个基于该状态的判断期间, 状态不会被并发修改, 从而使得这个检查操作相对于中断处理是原子的。
2. **防止数据不一致**: 如果不关闭中断, `still_alive` 可能读取到一个“脏”数据或中间状态。例如:
 - `still_alive` 开始读取进程 A 的状态, 发现是 `Ready` 。
 - 此时发生时钟中断, 调度器运行, 进程 A 可能因为某种原因 (如执行了 `exit` 系统调用, 其清理工作中断上下文中完成了一部分) 状态变为 `Dead` 。
 - 中断返回后, `still_alive` 基于之前读取的 (现在已过时的) `Ready` 状态返回 `true` , 而实际上进程 A 已经死亡。
3. **保护共享数据结构的操作**: 对进程列表或 PCB 的访问通常需要同步机制 (如锁)。关闭中断是一种简单而有效的临界区保护方法, 可以防止在访问这些共享结构时被中断打断, 从而避免了复杂的锁同步问题或潜在的死锁 (如果中断处理程序也试图获取相同的锁)。

3.2. 在不关闭中断的情况下, 会有什么问题?

1. **返回错误的结果**: `still_alive` 可能会返回不正确的信息。它可能报告一个已经死亡的进程仍然存活, 或者一个正在被销毁的进程状态不确定。
 2. **逻辑错误导致系统行为异常**: 依赖 `still_alive` 结果的内核逻辑 (例如 `wait` 函数) 可能会基于错误信息做出错误决策。如果 `wait` 函数错误地认为一个已退出的子进程仍然存活, 它可能会无限期等待, 导致父进程永久阻塞或系统资源无法释放。
 3. **潜在的系统崩溃**: 在极端情况下, 如果对共享数据结构 (如链表指针) 的不一致访问导致了内核数据结构的损坏, 可能会引发更严重的错误, 甚至导致系统崩溃 (Kernel Panic)。
- 因此, 在检查如进程存活状态这类敏感共享数据时, 关闭中断是一种必要的保护措施, 以确保操作的原子性和数据的一致性。

4. 对于如下程序, 使用 gcc 直接编译: …解释程序的运行。

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

程序的运行过程如下:

1. **编译与链接**:

- `gcc hello.c -o hello` :
- ▶ **预处理**: `#include <stdio.h>` 将标准输入输出库的声明包含进来。
- ▶ **编译**: C 代码被翻译成汇编代码。
- ▶ **汇编**: 汇编代码被转换成机器码, 生成目标文件 `hello.o`。此文件包含 `main` 函数的机器码和对外部函数 `printf` 的引用。
- ▶ **链接**: 链接器将 `hello.o` 与 C 标准库 (libc) 以及启动代码 (`_start`) 链接。 `printf` 的引用被解析为 libc 中的实际实现。最终生成可执行文件 `hello` (通常为 ELF 格式)。

2. 加载与进程创建 (以 Linux 为例):

- 用户在 shell 中执行 `./hello`。
- Shell (父进程) 调用 `fork()` 创建一个子进程。
- 子进程调用 `execve("./hello", ...)` 系统调用。
- **内核处理 `execve`** :
 - ▶ 内核验证 `hello` 文件的权限和格式 (ELF)。
 - ▶ 为新程序创建一个新的、干净的地址空间, 销毁子进程从 shell 继承的旧地址空间。
 - ▶ 根据 ELF 文件头中的程序头表 (Program Headers), 将代码段 (.text)、已初始化数据段 (.data) 从文件加载到内存的指定虚拟地址。为未初始化数据段 (.bss) 分配内存并清零。
 - ▶ 为进程设置用户栈, 并将命令行参数 (`argc`, `argv`) 和环境变量压栈。
 - ▶ 设置 CPU 寄存器, 特别是指令指针 (RIP) 指向程序的入口点 (ELF 文件中指定的 `_start` 地址), 栈指针 (RSP) 指向新设置的栈顶。

3. 执行:

- `execve` 返回到用户态, CPU 开始从 `_start` 执行。
- `_start` (由 C 运行时库提供) 进行初始化 (如设置 C 库环境), 然后调用 `main(argc, argv)`。
- `main` 函数执行:
 - ▶ `printf("Hello, World!\n");`
 - `printf` 是 libc 中的库函数。它格式化字符串。
 - 为了输出, `printf` 内部最终会调用 `write()` 系统调用。
 - **`write()` 系统调用过程**:
 1. `printf` 准备参数: 文件描述符 `STDOUT_FILENO` (通常为 1), 指向字符串 “Hello, World!\n” 的指针, 字符串长度。
 2. 执行 `syscall` 指令 (或 `int 0x80`), 陷入内核, 从用户态切换到内核态。
 3. 内核的系统调用分发器根据系统调用号找到 `sys_write`。
 4. `sys_write` 验证参数, 将数据从用户空间缓冲区复制到内核缓冲区。
 5. 内核通过文件描述符找到对应的输出设备驱动 (如 TTY 驱动)。
 6. 驱动程序将数据发送到终端显示。
 7. `sys_write` 返回写入的字节数。

8. 从内核态切换回用户态, `write()` 调用返回。

▸ `return 0;` : `main` 函数返回退出状态码 0。

4. 退出:

- `_start` 代码接收 `main` 的返回值, 并调用 `exit()` 系统调用, 参数为 0。
- `exit()` 系统调用过程:
 1. 陷入内核, 切换到内核态。
 2. 内核的 `sys_exit` 执行进程清理:
 - 关闭所有打开的文件描述符。
 - 释放进程占用的内存资源 (页表、物理页等)。
 - 将进程状态标记为 “僵尸” (Zombie)。
 - 保存退出状态码 (0)。
 - 向父进程 (shell) 发送 `SIGCHLD` 信号。
 3. 进程停止执行。其 PCB 和退出状态码保留, 直到父进程通过 `wait()` 或 `waitpid()` 回收。
- Shell 进程通过 `waitpid()` 获取到子进程的退出状态码, 内核彻底清理僵尸进程的资源。Shell 显示新的提示符。

5. x86_64::instructions::hlt 做了什么? 为什么这样使用? 为什么不可在用户态中的 `wait_pid` 实现中使用?

5.1. x86_64::instructions::hlt 做了什么?

`hlt` (Halt instruction) 是一条 x86/x86-64 架构的特权指令。当 CPU 执行 `hlt` 指令时, 它会:

1. 停止执行后续的指令。
2. 进入一种低功耗的暂停状态。

CPU 会保持这种暂停状态, 直到接收到以下任一事件:

- 一个外部硬件中断 (enabled interrupt)。
- 一个不可屏蔽中断 (NMI)。
- 一个调试异常。
- RESET 信号。

当中断发生后, 如果中断被启用 (`EFLAGS.IF = 1`), CPU 会处理该中断。中断处理程序返回后, CPU 通常会从 `hlt` 指令的下一条指令继续执行。

5.2. 为什么这样使用? (在实验代码的内核 `wait` 函数中)

在实验的 `ysos::wait` 函数中, 如果发现被等待的进程 (`init`) 仍然存活, 则执行 `hlt` :


```
pub fn wait(init: proc::ProcessId) {
    loop {
        if proc::still_alive(init) {
            x86_64::instructions::hlt(); // <--- 在此使用
        } else {
            break;
        }
    }
}
```

这样使用的主要原因：

1. **节能与效率**：如果 `init` 进程仍在运行，当前执行 `wait` 的内核代码（通常是系统空闲进程或引导过程中的某个等待点）没有其他紧急任务可做。与其在一个紧凑的 `loop {}` 中不断轮询 `still_alive`（这会浪费 CPU 周期并产生热量），不如执行 `hlt` 让 CPU 暂停。
2. **中断驱动的唤醒**：CPU 执行 `hlt` 后会暂停，直到下一次中断（通常是周期性的时钟中断）发生。时钟中断处理程序可能会运行调度器，或者其他事件可能导致 `init` 进程的状态改变（例如退出）。当中断处理完成后，CPU 会从 `hlt` 的下一条指令继续执行，再次进入 `loop`，重新调用 `still_alive` 检查 `init` 进程的状态。这形成了一种“中断驱动的轮询”，比纯粹的忙等待要高效得多。

5.3. 为什么不可以在用户态中的 `wait_pid` 实现中使用？

1. **特权指令限制**：`hlt` 是一条特权指令。它只能在内核态 (Ring 0) 执行。如果用户态程序 (Ring 3) 尝试执行 `hlt`，CPU 会产生一个通用保护故障 (#GP) 异常，导致用户程序崩溃。操作系统不允许用户程序直接控制 CPU 的核心状态，如暂停整个处理器。
2. **用户态应使用阻塞式系统调用**：用户态程序等待子进程结束应该使用操作系统提供的标准系统调用，如 `wait()` 或 `waitpid()`。
 - 当用户程序调用 `waitpid()` 时，它会陷入内核。
 - 内核中的 `sys_waitpid` 会检查目标子进程的状态。
 - 如果子进程尚未退出，内核会将调用 `waitpid` 的父进程标记为“阻塞” (Blocked) 或“睡眠” (Sleeping) 状态，并将其从调度器的就绪队列中移除。
 - 然后，内核会调用调度器，选择另一个就绪的进程来运行。这样，等待的父进程就不会消耗 CPU 时间。
 - 当子进程退出时，内核会唤醒父进程（将其状态改回“就绪”并放回就绪队列）。当父进程再次被调度运行时，其 `waitpid` 系统调用就可以返回子进程的退出信息。
3. **系统稳定性与公平性**：如果允许用户程序随意执行 `hlt`，可能会导致整个系统无响应或行为不可预测。例如，一个用户程序 `hlt` 后，如果没有正确的唤醒机制，它可能永久暂停，也可能阻止其他进程（包括重要的系统进程）运行。操作系统的调度器负责公平地分配 CPU 时间给所有活动进程。

总结：`hlt` 是内核在确定没有即时任务时暂停 CPU 以节能并等待中断的有效手段。用户态程序则依赖操作系统提供的阻塞式系统调用（如 `waitpid`）来实现等待，由内核负责管理进程状态和调度，而不是让用户程序直接使用特权指令控制 CPU。

6. 有同学在某个回南天迷蒙的深夜遇到了奇怪的问题：…尝试解释这个现象。

这个现象——仅在串口输入中断时触发 Page Fault，而其他如进程切换、内存分配、fork 等系统调用正常，且问题根源在于 TSS 中的 `privilege_stack_table`（即 IST, Interrupt Stack Table）相关设置被注释掉——可以如下解释：

6.1. 特权级栈 (IST) 的作用

在 x86-64 架构中，TSS (Task State Segment) 包含一个 IST。IST 允许为特定的中断或异常（如 Page Fault, Double Fault, NMI，以及其他可配置的硬件中断）预定义独立的栈。当这些特定的中断或异常发生，并且需要从低特权级（如用户态 Ring 3）切换到高特权级（内核态 Ring 0）时，CPU 会自动切换到 IST 中为该中断/异常向量指定的栈，而不是使用当前任务的常规 Ring 0 栈 (RSP0)。

主要优点：

1. **处理栈溢出或栈损坏**：如果当前栈（用户栈或当前内核栈）已满或损坏，此时发生中断/异常，若试图在原栈上操作，可能导致双重故障甚至三重故障（系统重置）。IST 提供了一个干净、可靠的栈空间，确保关键的错误处理程序能够运行。
2. **处理特定中断的栈需求**：某些中断处理可能需要较大的栈空间或有特殊对齐要求。

6.2. 中断触发与栈切换过程（以串口中断为例）

1. 用户程序在 Ring 3 运行。
2. 串口输入，硬件触发中断请求 (IRQ)。
3. CPU 响应中断，通过 IDT 找到对应的门描述符。
4. 发生特权级切换 (Ring 3 -> Ring 0)。
5. **栈切换**：
 - 如果 IDT 门描述符中为该中断指定了 IST 索引 (一个非零值)：CPU 会从 TSS 的 IST 中加载该索引对应的栈指针 (RSP) 和栈段 (SS)，切换到这个 IST 栈。
 - 如果未指定 IST 索引 (索引为 0)：CPU 会从 TSS 中加载常规的 Ring 0 栈指针 (RSP0) 和栈段 (SS0)。
6. CPU 将旧的 SS, RSP, RFLAGS, CS, RIP (以及可能的错误码) 压入新的内核栈。
7. CPU 加载中断处理程序的 CS 和 RIP，开始执行中断服务例程 (ISR)。

6.3. 解释现象

1. **TSS 中 `privilege_stack_table` (IST) 设置被注释掉**: 这意味着 TSS 中 IST 表的条目可能未被正确初始化, 或者指向了无效的内存地址 (例如, 全零, 或者指向一个未映射、不可写的区域)。
 2. **为什么只有串口输入中断触发 Page Fault**:
 - **串口中断配置**: 很可能操作系统的 IDT 中, 为串口中断配置了使用某个 IST 条目 (例如, IST1)。
 - **中断发生时的栈切换尝试**: 当串口中断发生, CPU 尝试从 Ring 3 切换到 Ring 0, 并根据 IDT 的指示去 TSS 的 IST 中查找 IST1 的栈指针。
 - **无效的 IST 栈指针**: 由于 IST 相关设置被注释, TSS 中 IST1 的条目可能是 0x0, 或者指向一个非法的、未映射的地址。
 - **写入非法地址导致 Page Fault**: CPU 获取了这个非法的栈指针后, 下一步是在这个“新栈”上压入旧的 SS, RSP, RFLAGS, CS, RIP。如果这个栈指针是 0x0 或指向一个不可写的内存区域, 那么第一次写操作 (压栈) 就会访问非法地址, 从而立即触发一个 Page Fault (0x0E)。
 3. **为什么其他操作 (进程切换、内存分配、fork) 正常**:
 - **系统调用 (fork, 内存分配等)**: 系统调用通过 `syscall` 或 `int 0x80` 等指令进入内核。它们通常使用 TSS 中为 Ring 0 定义的常规栈 (SS0, RSP0)。只要这个常规内核栈是正确设置的 (通常在创建任务或上下文切换时会设置好), 这些系统调用就能正常工作。它们不一定依赖于特定的 IST 条目。
 - **进程切换**: 进程切换由调度器在内核态完成。调度器会保存当前进程的上下文 (包括其内核栈指针) 并加载下一个进程的上下文 (包括其内核栈指针)。这个过程直接操作已知的、合法的内核栈, 不涉及通过 IDT 和 IST 查找新栈的过程。
 4. **“注意到一个不应当存在的地址……?”**:
 - 这个“不应当存在的地址”很可能就是导致 Page Fault 的地址, 记录在 CR2 寄存器中。
 - 如果 IST 条目是 0, 那么 CPU 可能会尝试向地址 0x0 附近压栈, CR2 就会是一个非常小的值。
 - Page Fault 的错误码也会提供线索:
 - P (Present) 位 = 0: 页面不存在。
 - W/R (Write/Read) 位 = 1: 尝试写入。
 - U/S (User/Supervisor) 位 = 0: 错误发生在内核态。这三者结合, 完美符合内核在尝试向一个由错误的 IST 条目提供的无效地址 (作为栈顶) 写入数据 (压入中断栈帧) 时发生的情况。
- 验证方法**: 使用调试工具 (如 GDB 配合 QEMU, 或实验提供的 `ysos.py -i`) 捕获中断和异常信息:
1. 触发串口输入。
 2. 观察中断序列: 应先看到串口中断向量。
 3. 紧接着, 应看到 Page Fault (0x0E)。

4. 检查 Page Fault 时的 CR2 寄存器值（应为非法地址，如 0 或接近 0）。
5. 检查 Page Fault 的错误码（应指示内核态写一个不存在的页面）。
6. 检查 TSS dump，确认被串口中断使用的 IST 条目确实是 0 或指向无效区域。

结论：当为特定中断（如串口中断）配置了使用 IST，但 TSS 中对应的 IST 条目未被正确初始化（例如被注释掉导致其值为 0 或无效地址）时，中断发生后 CPU 在尝试切换到这个无效的 IST 栈并压入栈帧时，会因访问非法内存而触发 Page Fault。其他不依赖此特定 IST 条目的内核操作则可能正常运行。