中山大學

SUN YAT-SEN UNIVERSITY

实验报告

实验四：用户程序与系统调用

姓　　名	元朗曦
学　　号	23336294
班　　级	计算机八班
专　　业	计算机科学与技术
学　　院	计算机学院

1 合并实验代码

在 pkg/app 中，定义提供了一些用户程序，这些程序将会在编译后提供给内核加载运行。

在 pkg/syscall 中，提供系统调用号和调用约束的定义，将会在内核和用户库中使用，在下文中会详细介绍。

在 pkg/lib 中，定义了用户态库并提供了一些基础实现，相关内容在下文中会详细介绍。

在 pkg/kernel 中，添加了如下一些模块：

- interrupt/syscall：定义系统调用及其服务的实现。
- memory/user：用户堆内存分配的实现，会被用在系统调用的处理中，将用户态的内存分配委托给内核。
- utils/resource：定义了用于进行 I/O 操作的 Resource 结构体，用于处理用户态的读写系统调用。

2 用户程序

2.1 编译用户程序

对于不同的运行环境，即使指令集相同，一个可执行的程序仍然有一定的差异。

与内核的编译类似，在 pkg/app/config 中，定义了用户程序的编译目标，并定义了相关的 LD 链接脚本。

在 Cargo.toml 中，使用通配符引用了 pkg/app 中的所有用户程序。相关的编译过程在先前给出的编译脚本中均已定义，可以直接编译。

通常而言，用户程序并不直接自行处理系统调用，而是由用户态库提供的函数进行调用。

为了让用户态程序更好地与 YSOS 进行交互，处理程序的生命周期，便于编写用户程序等，我们需要提供用户态库，以便用户程序调用。

用户态库被定义在 pkg/lib 中，在用户程序中，编辑 Cargo.toml，使用如下方式引用用户库：

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

一个简单的用户程序示例如下所示，同样存在于 app/hello/src/main.rs 中：

```rust
#![no_std]
#![no_main]

use lib::*;

extern crate lib;

fn main() -> isize {
    println!("Hello, world!!!");

    233
}
```

```
13
14 entry!(main);
```

其中：

- `#![no_std]` 表示不使用标准库，Rust 并没有支持 YSOS 的标准库，需要我们自行实现。
- `#![no_main]` 表示不使用标准的 `main` 函数入口，而是使用 `entry!` 宏定义的入口函数。

2.2 加载程序文件

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。

目前的内核尚不具备访问磁盘和文件系统，并将它们读取加载的能力（将会在实验六中实现），因此需要另辟蹊径：在 bootloader 中将符合条件的用户程序加载到内存中，并将它们交给内核，用于生成用户进程。

为了存储用户程序的相关信息，我们在 `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：

```rust
/// App information
pub struct App<'a> {
    /// The name of app
    pub name: ArrayString<16>,
    /// The ELF file
    pub elf: xmas_elf::ElfFile<'a>,
}

pub type AppList = ArrayVec<App<'static>, 16>;
pub type AppListRef = Option<&'static AppList>;

/// This structure represents the information that the bootloader passes to the kernel.
pub struct BootInfo {
    /// The memory map
    pub memory_map: MemoryMap,

    /// The offset into the virtual address space where the physical memory is mapped.
    pub physical_memory_offset: u64,

    /// The system table virtual address
    pub system_table: NonNull<core::ffi::c_void>,

    /// Loaded apps
    pub loaded_apps: Option<AppList>,
}
```

之后，我们在 `pkg/boot/src/fs.rs` 中，创建函数 `load_apps` 用于加载用户程序：

```rust
/// Load apps into memory, when no fs implemented in kernel
///
```

```rust
/// List all file under "APP" and load them.
pub fn load_apps() -> AppList {
    let mut root = open_root();
    let mut buf = [0; 8];
    let cstr_path = uefi::CStr16::from_str_with_buf("\\APP\\", &mut buf).unwrap();

    let mut handle = root
        .open(cstr_path, FileMode::Read, FileAttribute::empty())
        .unwrap()
        .into_directory()
        .expect("Failed to open app directory");

    let mut apps = ArrayVec::new();
    let mut entry_buf = [0u8; 0x100];

    loop {
        let info = handle
            .read_entry(&mut entry_buf)
            .expect("Failed to read entry");

        match info {
            Some(entry) => {
                let file = handle
                    .open(entry.file_name(), FileMode::Read, FileAttribute::empty())
                    .expect("Failed to open file");

                if file.is_directory().unwrap_or(true) {
                    continue;
                }

                let elf = {
                    // DONE: load file with `load_file` function
                    // DONE: convert file to `ElfFile`
                    let mut file = file.into_regular_file().unwrap();
                    let buf = load_file(&mut file);
                    ElfFile::new(buf).expect("Failed to parse ELF file")
                };

                let mut name = ArrayString::<16>::new();
                entry.file_name().as_str_in_buf(&mut name).unwrap();

                apps.push(App { name, elf });
            }
            None => break,
```

```
        }
    }

    info!("Loaded {} apps", apps.len());

    apps
}
```

在 pkg/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：

```rust
#[uefi::entry]
fn boot_main() -> uefi::Status {
    // ...

    // 3. Load apps
    let apps = if config.load_apps {
        info!("Loading apps...");
        Some(load_apps())
    } else {
        info!("Skip loading apps");
        None
    };

    // ...

    // construct BootInfo
    let bootinfo = BootInfo {
        memory_map: mmap.entries().copied().collect(),
        physical_memory_offset: config.physical_memory_offset,
        system_table,
        loaded_apps: apps,
    };
}
```

修改 ProcessManager 的定义与初始化逻辑，将 AppList 添加到 ProcessManager 中：

```rust
pub fn init(init: Arc<Process>, app_list: AppListRef) {
    // DONE: set init process as Running
    init.write().resume();
    // DONE: set processor's current pid to init's pid
    processor::set_pid(init.pid());

    PROCESS_MANAGER.call_once(|| ProcessManager::new(init ,app_list));
}

// ...
```

```rust

pub struct ProcessManager {
    processes: RwLock<BTreeMap<ProcessId, Arc<Process>>>,
    ready_queue: Mutex<VecDeque<ProcessId>>,
    app_list: AppListRef,
}

impl ProcessManager {
    pub fn new(init: Arc<Process>, app_list: AppListRef) -> Self {
        let mut processes = BTreeMap::new();
        let ready_queue = VecDeque::new();
        let pid = init.pid();

        trace!("Init {:#?}", init);

        processes.insert(pid, init);
        Self {
            processes: RwLock::new(processes),
            ready_queue: Mutex::new(ready_queue),
            app_list,
        }
    }

    // ...
}
```

之后修改 `pkg/kernel/src/proc/mod.rs`，将 `app_list` 传递给 `init`：

```rust
/// init process manager
pub fn init(boot_info: &'static BootInfo) {
    let proc_vm = ProcessVm::new(PageTableContext::new()).init_kernel_vm();

    trace!("Init kernel vm: {:#?}", proc_vm);

    // kernel process
    let kproc = Process::new(String::from("kernel"), None, Some(proc_vm), None);
    let app_list = boot_info.loaded_apps.as_ref();
    manager::init(kproc, app_list);

    info!("Process Manager Initialized.");
}
```

在 `pkg/kernel/src/proc/mod.rs` 中，定义一个 `list_app` 函数，用于列出当前系统中的所有用户程序和相关信息：

```rust
pub fn list_app() {
    x86_64::instructions::interrupts::without_interrupts(|| {
```

```
        let Some(app_list) = get_process_manager().app_list() else {
            warn!("No app found in list!");
            return;
        };

        let apps = app_list
            .iter()
            .map(|app| app.name.as_str())
            .collect::<Vec<&str>>()
            .join(", ");

        // TODO: print more information like size, entry point, etc.

        info!("App list: {}", apps);
    });
}
```

2.3 创建用户进程

在 `pkg/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：

```Rust
pub fn spawn(name: &str) -> Option<ProcessId> {
    let app = x86_64::instructions::interrupts::without_interrupts(|| {
        let app_list = get_process_manager().app_list()?;
        app_list.iter().find(|&app| app.name.eq(name))
    })?;

    elf_spawn(name.to_string(), &app.elf)
}

pub fn elf_spawn(name: String, elf: &ElfFile) -> Option<ProcessId> {
    let pid = x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        let process_name = name.to_lowercase();
        let parent = Arc::downgrade(&manager.current());
        let pid = manager.spawn(elf, name, Some(parent), None);

        debug!("Spawned process: {}#{}", process_name, pid);
        pid
    });

    Some(pid)
}
```

在后续的实验中，`spawn` 将接收一个文件路径，操作系统需要从文件系统中读取文件，并将其加载到内存中。通过将 `elf_spawn` 独立出来，我们可以在后续实验中直接对接到文件系统的读取结果，而无需修改后续代码。

在 pkg/kernel/src/proc/manager.rs 中，实现 spawn 函数：

```rust
impl ProcessManager {
    // ...

    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);

        let mut inner = proc.write();
        // DONE: load elf to process pagetable
        // DONE: alloc new stack for process
        // DONE: mark process as ready
        inner.pause();
        inner.load_elf(elf);
        inner.init_stack_frame(
            VirtAddr::new(elf.header.pt2.entry_point()),
            VirtAddr::new(super::stack::STACK_INIT_TOP),
        );
        drop(inner);

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

        let pid = proc.pid();
        // DONE: something like kernel thread
        self.add_proc(pid, proc);
        self.push_ready(pid);

        pid
    }

    // ...
}
```

之后在 ProcessInner 和 ProcessVm 中实现 load_elf 函数，来处理代码段映射等内容，

```rust
impl ProcessInner {
    // ...

```

```
    pub fn load_elf(&mut self, elf: &ElfFile) {
        self.vm_mut().load_elf(elf);
    }

    // ...
}
```

```Rust
impl ProcessVm {
    // ...

    pub fn load_elf(&mut self, elf: &ElfFile) {
        let mapper = &mut self.page_table.mapper();
        let alloc = &mut *get_frame_alloc_for_sure();

        self.stack.init(mapper, alloc);
        self.load_elf_code(elf, mapper, alloc);
    }
    fn load_elf_code(&mut self, elf: &ElfFile, mapper: MapperRef, alloc: FrameAllocatorRef) {
        elf::load_elf(elf, *PHYSICAL_OFFSET.get().unwrap(), mapper, alloc, true).unwrap();
    }

    // ...
}
```

并修改 `pkg/elf/src/lib.rs` 中 `load_elf` 函数的实现，使其能处理用户权限要求。

```Rust
/// Load & Map ELF file
///
/// for each segment, load code to new frame and set page table
pub fn load_elf(
    elf: &ElfFile,
    physical_offset: u64,
    page_table: &mut impl Mapper<Size4KiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
    user_access: bool,
) -> Result<Vec<PageRangeInclusive>, MapToError<Size4KiB>> {
    trace!("Loading ELF file...{:?}", elf.input.as_ptr());
    elf.program_iter()
        .filter(|segment| segment.get_type().unwrap() == program::Type::Load)
        .map(|segment| {
            load_segment(
                elf,
                physical_offset,
                &segment,
                page_table,
```

```
                frame_allocator,
                user_access,
            )
        })
        .collect()
}
```

同时，需要在 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。补充 `pkg/kernel/src/memory/gdt.rs` 如下：

```Rust
lazy_static! {
    static ref GDT: (GlobalDescriptorTable, KernelSelectors, UserSelectors) = {
        // ...

        let user_code_selector = gdt.append(Descriptor::user_code_segment());
        let user_data_selector = gdt.append(Descriptor::user_data_segment());
        (
            // ...

            UserSelectors {
                user_code_selector,
                user_data_selector,
            },
        )
    };
}
```

之后将其通过合适的方式暴露出来，以供栈帧初始化时使用：

```Rust
impl ProcessContext {
    // ...

    pub fn init_stack_frame(&mut self, entry: VirtAddr, stack_top: VirtAddr) {
        self.value.stack_frame.stack_pointer = stack_top;
        self.value.stack_frame.instruction_pointer = entry;
        self.value.stack_frame.cpu_flags =
            RFlags::IOPL_HIGH | RFlags::IOPL_LOW | RFlags::INTERRUPT_FLAG;

        //let selector = get_selector();
        //self.value.stack_frame.code_segment = selector.code_selector;
        //self.value.stack_frame.stack_segment = selector.data_selector;
        let selector = get_user_selector();
        self.value.stack_frame.code_segment = selector.user_code_selector;
        self.value.stack_frame.stack_segment = selector.user_data_selector;

        trace!("Init stack frame: {:#?}", &self.stack_frame);
    }
```

```
19 }
```

3 系统调用的实现

为了为用户程序提供服务，操作系统需要实现一系列的系统调用，从而为用户态程序提供内核态服务。这些操作包括文件操作、进程操作、内存操作等，相关的指令一般需要更高的权限（相对于用户程序）才能执行。

3.1 调用约定

系统调用一般有系统调用号、参数、返回值等调用约定，不同的上下文参数对应的系统调用的行为存在不同。

以 x86_64 的 Linux 为例，系统调用的部分调用约定如下所示：

- 系统调用号通过 `rax` 寄存器传递
- 参数通过 `rdi`、`rsi`、`rdx`、`r10`、`r8`、`r9` 寄存器传递
- 参数数量大于 6 时，通过栈传递
- 返回值通过 `rax` 寄存器传递

这里的读写、进程操作的系统调用号基本与 Linux 中功能类似的系统调用号一致，而有些系统调用号则是自定义的。

- `ListApp` 用于列出当前系统中的所有用户程序，由于尚不会进行文件系统的实现，因此需要这样一个系统调用来获取用户程序的信息。
- `Stat` 用于获取系统中的一些统计信息，例如内存使用情况、进程列表等，用于调试和监控。
- `Allocate/Deallocate` 用于分配和释放内存。在当前没有完整的用户态内存分配支持的情况下，可以利用系统调用将其委托给内核来完成。

3.2 软中断处理

在 `pkg/kernel/src/interrupt/syscall/mod.rs` 中，补全中断注册函数，

```rust
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    // DONE: register syscall handler to IDT
    //       - standalone syscall stack
    //       - ring 3
    unsafe {
        idt[consts::Interrupts::Syscall as u8]
            .set_handler_fn(syscall_handler)
            .set_stack_index(gdt::SYSCALL_IST_INDEX)
            .set_privilege_level(x86_64::PrivilegeLevel::Ring3);
    }
}
```

并在 `pkg/kernel/src/interrupt/mod.rs` 中调用它：

```rust
lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
```

```
        unsafe {
            exceptions::register_idt(&mut idt);
            clock::register_idt(&mut idt);
            serial::register_idt(&mut idt);
            syscall::register_idt(&mut idt);
        }
        idt
    };
}
```

4 用户态库的实现

用户态库是用户程序的基础，它提供了一些基础的函数，用于调用系统调用，实现一些基础的功能。

在这一部分的实现中，我们着重实现了 `read` 和 `write` 系统调用的封装和内核侧的实现，并通过内存分配、释放的系统调用，给予用户态程序动态内存分配的能力。

4.1 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，我们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。

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。为了调试和安全性考量，这部分内存还需要 `NO_EXECUTE` 标志位。

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。

补全代码，实现用户堆的初始化：

```Rust
pub fn init_user_heap() -> Result<(), MapToError<Size4KiB>> {
    // Get current pagetable mapper
    let mapper = &mut PageTableContext::new().mapper();
    // Get global frame allocator
    let frame_allocator = &mut *super::get_frame_alloc_for_sure();

    // DONE: use elf::map_range to allocate & map
    //       frames (R/W/User Access)
    let page_range = {
        let heap_start = VirtAddr::new(USER_HEAP_START as u64);
        let heap_start_page = Page::containing_address(heap_start);
        let heap_end_page = heap_start_page + USER_HEAP_PAGE as u64 - 1u64;
        Page::range(heap_start_page, heap_end_page)
    };

    debug!(
        "User Heap        : 0x{:016x}-0x{:016x}",
```

```
        page_range.start.start_address().as_u64(),
        page_range.end.start_address().as_u64()
    );

    let (size, unit) = crate::humanized_size(USER_HEAP_SIZE as u64);
    info!("User Heap Size   : {:>7.*} {}", 3, size, unit);

    for page in page_range {
        let frame = frame_allocator
            .allocate_frame()
            .ok_or(MapToError::FrameAllocationFailed)?;
        let flags =
            PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
            PageTableFlags::USER_ACCESSIBLE;
        unsafe { mapper.map_to(page, frame, flags, frame_allocator)?.flush() };
    }

    unsafe {
        USER_ALLOCATOR
            .lock()
            .init(USER_HEAP_START as *mut u8, USER_HEAP_SIZE);
    }

    Ok(())
}
```

4.2 标准输入输出

为了在系统调用中实现基础的读写操作，代码中定义了一个 `Resource` 枚举，并借用 Linux 中“文件描述符”的类似概念，将其存储在进程信息中。

在 `pkg/kernel/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”；在 `ProcessData` 的 `default` 函数中初始化，添加默认的资源：

```rust
#[derive(Debug, Clone)]
pub struct ProcessData {
    // shared data
    pub(super) env: Arc<RwLock<BTreeMap<String, String>>>,
    pub(super) resources: Arc<RwLock<ResourceSet>>,
}

impl Default for ProcessData {
    fn default() -> Self {
        Self {
            env: Arc::new(RwLock::new(BTreeMap::new())),
            resources: Arc::new(RwLock::new(ResourceSet::default())),
        }
    }
```

```rust
}

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

    pub fn env(&self, key: &str) -> Option<String> {
        self.env.read().get(key).cloned()
    }

    pub fn set_env(&mut self, key: &str, val: &str) {
        self.env.write().insert(key.into(), val.into());
    }

    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 `pkg/kernel/src/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：

```rust
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))
}
```

补全 `pkg/kernel/src/interrupt/syscall/service.rs` 中的 `syscall_write` 函数，用于处理 `write` 系统调用：

```rust
pub fn sys_write(args: &SyscallArgs) -> usize {
    // DONE: get buffer and fd by args
    //      - core::slice::from_raw_parts
    // DONE: call proc::write -> isize
    // DONE: return the result as usize
    let buf = match as_user_slice(args.arg1, args.arg2) {
        Some(buf) => buf,
        None => return usize::MAX,
```

```
    };

    let fd = args.arg0 as u8;
    write(fd, buf) as usize
}
```

sys_read 的实现与 sys_write 类似：

```rust
pub fn sys_read(args: &SyscallArgs) -> usize {
    // DONE: just like sys_write
    let buf = match as_user_slice_mut(args.arg1, args.arg2) {
        Some(buf) => buf,
        None => return usize::MAX,
    };

    let fd = args.arg0 as u8;
    read(fd, buf) as usize
}
```

4.3 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。

由于此时通过中断进入内核态，与时钟中断类似，操作系统得以控制退出中断时的 CPU 上下文。因此可以在退出的时候清理进程占用的资源，并调用 switch_next 函数，切换到下一个就绪的进程。

在 pkg/kernel/src/proc/mod.rs 中实现 process_exit 函数，封装对应的功能，并暴露给系统调用：

```rust
pub fn process_exit(ret: isize, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        // DONE: implement this for ProcessManager
        manager.kill_current(ret);
        manager.switch_next(context);
    })
}
```

我们修改用户态库中的 entry! 和 panic 函数，在用户程序中调用 exit 系统调用，并传递一个返回值，以验证用户程序的退出功能：

```rust
#[macro_export]
macro_rules! entry {
    ($fn:ident) => {
        #[unsafe(export_name = "_start")]
        pub extern "C" fn __impl_start() {
            let ret = $fn();
            // DONE: after syscall, add lib::sys_exit(ret);
```

```
            sys_exit(ret);
        }
    };
}

#[cfg_attr(not(test), panic_handler)]
fn panic(info: &core::panic::PanicInfo) -> ! {
    let location = if let Some(location) = info.location() {
        alloc::format!(
            "{}@{}:{}",
            location.file(),
            location.line(),
            location.column()
        )
    } else {
        "Unknown location".to_string()
    };

    errln!(
        "\n\n\rERROR: panicked at {}\n\n\r{}",
        location,
        info.message()
    );

    // DONE: after syscall, add lib::sys_exit(1);
    crate::sys_exit(1)
}
```

系统应正常创建 `hello` 进程，输出 `Hello, world!!!`，并正确退出。

5 运行 Shell

至此，我们已经可以编写一个简单的 Shell 了。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

创建 `pkg/app/sh` 包，用于实现 Shell 的功能。在 `Cargo.toml` 中添加依赖：

TOML

```
[package]
name = "ysos_sh"
version.workspace = true
edition.workspace = true

[dependencies]
lib.workspace = true
```

在 src/main.rs 中，借助系统调用实现 Shell 的基本功能：

```rust
#![no_std]
#![no_main]

use lib::*;
extern crate lib;

use alloc::vec::Vec;

fn main() -> isize {
    loop {
        print!("> ");

        let input = stdin().read_line();
        let args: Vec<&str> = input.split_whitespace().collect();

        if args.is_empty() {
            continue;
        }

        match args[0] {
            "exit" | "\x04" => {
                break;
            },
            "exec" => {
                if args.len() == 1 {
                    println!("Usage: exec <app>");
                    continue;
                }

                let name = args[1];
                let pid = sys_spawn(name);
                let ret = sys_wait_pid(pid);
                println!("Process {}#{} exited with code {}", name, pid, ret);
            },
            "help" => {
                println!("Available commands:");
                println!("  exec <app>        Execute an application");
                println!("  exit              Exit the shell");
                println!("  help              Show this help message");
                println!("  list apps|proc    List all applications or
                processes");
            },
            "list" => {
                if args.len() == 1 {
                    println!("Usage: list apps|proc");
```

```
                    continue;
                }

                match args[1] {
                    "apps" => sys_list_app(),
                    "proc" => sys_stat(),
                    _ => println!("Usage: list apps|proc"),
                }
            }
            _ => {
                println!("Command not found: {}", args[0]);
            },
        }
    }

    0
}

entry!(main);
```

我们在 pkg/app/fact 包中实现如下测试程序：

```rust
const MOD: u64 = 1000000007;

fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n - 1) % MOD
    }
}

fn main() -> isize {
    print!("Input n: ");

    let input = lib::stdin().read_line();

    // prase input as u64
    let n = input.parse::<u64>().unwrap();

    if n > 1000000 {
        println!("n must be less than 1000000");
        return 1;
    }

    // calculate factorial
```

```
25      let result = factorial(n);
26
27      // print system status
28      sys_stat();
29
30      // print result
31      println!("The factorial of {} under modulo {} is {}.", n, MOD, result);
32
33      0
34  }
35
36  entry!(main);
```

系统启动后会自动运行 `sh`，输入 `exec fact`，测试程序应正常运行，请求输入 `n`，并输出 `n` 的阶乘。

6 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。

> **解答**
>
> 在内核线程中不能直接使用系统调用，因为系统调用是为用户态进程设计的，而内核线程运行在内核态，与用户态进程有本质的区别。
>
> 系统调用是用户态程序通过特定的接口请求操作系统内核服务的一种机制。它通过 `trap` 指令从用户态切换到内核态，内核处理后再返回用户态。由于内核线程已经在内核态运行，因此系统调用在内核线程中没有意义，且不会按照预期工作。

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。

> **解答**
>
> 克隆内核页表是为了实现用户态和内核态的隔离，同时提供内核空间的共享访问能力。
>
> 1. 隔离用户态和内核态：
>   - 用户态程序只能访问用户空间的虚拟内存，不能直接访问内核空间。这种隔离是通过页表权限控制实现的。
>   - 内核态需要能够访问所有的内存（包括用户空间和内核空间），因此需要页表覆盖整个内存范围。
> 2. 共享内核空间：
>   - 每个进程的页表需要映射相同的内核空间（通常是高地址部分），以便在执行系统调用或进入内核态时，能够正确访问内核内存。
>   - 克隆内核页表可以避免为每个进程重复创建内核空间的映射，从而节省内存。
> 3. 性能优化：
>   - 克隆内核页表减少了页表创建和管理的开销，因为内核空间的映射是静态的，不会随进程切换或用户态内存变化而变化。
>
> 系统调用的页表：
>
> - 在执行系统调用时，处理器仍然使用当前进程的页表（即用户态页表），因为用户空间和内核空间是共享的。
> - 进程的页表中包含对用户空间和内核空间的映射：
>   - 用户空间部分的映射具有用户态访问权限（U 位设置）。
>   - 内核空间部分的映射具有内核态访问权限（U 位未设置）。
>
> 用户态程序尝试访问内核空间时，由于页表中内核空间的映射未设置用户态权限，会触发处理器的保护机制，产生 `segfault` 或 `page fault`。 这种机制确保用户态程序无法直接访问内核空间。

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？

> **解答**
>
> 如果在调用 `still_alive` 这类函数时不先关闭中断，可能会出现以下问题：
>
> 1. 临界区打断：当你在检查进程状态的时候，如果中断未被关闭，可能会在状态尚未被完全读取前被中断打断。中断处理程序可能会修改进程状态（例如在定时器中更新或者切换进程时修改状态），导致读取到的状态是不一致的。
> 2. 竞争条件：没有关闭中断容易导致多个硬件中断或其他执行路径在同一时间段内访问同一个数据结构，从而引起竞争条件。这会破坏数据的一致性，使得判断错误（例如误判进程已终止或仍存活）。

4. 对于如下程序，使用 `gcc` 直接编译：

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

从本次实验及先前实验的所学内容出发，结合进程的创建、链接、执行、退出的生命周期，参考系统调用的调用过程（可以仅以 Linux 为例），解释程序的运行。

> **解答**
>
> 1. 编译与链接
>
>   编译器进行以下步骤：
>
>   - 预处理：处理 `#include` 指令，展开宏等。
>   - 编译：将 C 代码编译为汇编代码。
>   - 汇编：将汇编代码转换为机器语言生成目标文件（`*.o` 文件）。
>   - 链接：将目标文件与 C 库（如 `libc`）以及其他依赖的库链接，生成可执行文件。
>
> 2. 程序执行与进程生命周期
>
> 在命令行输入 `./hello` 时，shell 会执行以下步骤：
>
> - 创建一个新的进程（`fork`），并在子进程中执行 `execve` 系统调用，加载可执行文件。
> - 内核加载 ELF 文件，设置进程的页表、栈、堆等。
> - 内核将控制权转移到用户程序的入口点（`main` 函数）。
> - 用户程序开始执行，调用 `libc` 中对 `printf` 的实现。
> - 当 `main` 函数返回时，程序调用 `exit` 系统调用，通知内核进程结束。

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

> **解答**
>
> `x86_64::instructions::hlt` 是一个封装了 x86 架构下 HLT 汇编指令的函数，其主要功能是让处理器进入低功耗的空闲状态，直到下一个硬件中断发生。
>
> 1. 在内核态中使用 `hlt` 可以节省 CPU 能耗，同时也避免 CPU 轮询对系统资源的浪费。
> 2. `hlt` 指令属于特权指令，只允许在内核态下执行。如果用户程序尝试执行 `hlt`，会触发违反指令执行权限的异常。

6. 有同学在某个回南天迷蒙的深夜遇到了奇怪的问题：

   只有当进行用户输入（触发了串口输入中断）的时候，会触发奇怪的 Page Fault，然而进程切换、内存分配甚至 `fork` 等系统调用都很正常。

经过~~近三个小时~~的排查，发现他将 TSS 中的 `privilege_stack_table` 相关设置注释掉了。

请查阅资料，了解特权级栈的作用，实验说明这一系列中断的触发过程，尝试解释这个现象。

- 可以使用 `intdbg` 参数，或 `ysos.py -i` 进行数据捕获。
- 留意 `0x0e` 缺页异常和缺页之前的中断的信息。
- 注意到一个不应当存在的地址……？

或许你可以重新复习一下 Lab 2 的相关内容：double-fault-exceptions

解答

在 x86_64 架构中，当 CPU 从低特权级（比如用户态 ring3）切换到高特权级（比如内核态 ring0）时，硬件需要切换到一个专门的内核栈。这个内核栈的地址就是在 TSS（任务状态段）中的 `privilege_stack_table` 中指定的。这样做可以防止用户态和内核态使用同一套栈，从而提高安全性和稳定性。

当串口接收到用户输入时，会触发一个硬件中断。当 CPU 判断当前特权级与中断目标特权级不同时（通常从 ring3 到 ring0），硬件会查找 TSS 中对应特权级（一般是 ring0）的栈指针（RSP0），并将其加载到栈寄存器中，以便后续中断处理过程中使用一个预先分配的内核栈。

如果将 TSS 中关于 `privilege_stack_table` 的设置注释掉，那么 TSS 中并没有正确配置内核态栈的地址。这样当发生中断——特别是从用户态切换到内核态的时候——CPU 无法找到合法的内核栈指针，就可能会使用一个未被正确映射的地址或是“脏数据”，从而触发 Page Fault。像 `fork`、内存分配、进程切换等系统调用因为在进入内核态时已经通过其它机制设置了合适的栈所以不会出现这种问题。