



# 课程实验报告

## 0x04 用户程序与系统调用

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 5 月 6 日

# 目录

<b>1 代码实现 .....</b>	<b>3</b>
1.1 加载程序文件 .....	3
1.2 生成用户程序 .....	5
1.3 系统调用的实现 .....	8
1.4 用户态库的实现 .....	10
1.5 进程退出 .....	12
1.6 进程创建与等待 .....	13
1.7 Shell 的简单实现 .....	13
<b>2 思考题 .....</b>	<b>15</b>
2.1 q1 .....	15
2.2 q2 .....	15
2.3 q3 .....	17
2.4 q4 .....	17
2.5 q5 .....	18
2.6 q6 .....	19
<b>3 实验结果展示 .....</b>	<b>19</b>

# 1 代码实现

## 1.1 加载程序文件

### pkg/boot/src/lib.rs

存储用户程序相关信息。

```
1  pub type MemoryMap = ArrayVec<MemoryDescriptor, 256>;  
2  
3  pub const APP_LIST_MAX: usize = 16;  
4  pub struct App {  
5      pub name: ArrayString<16>,  
6      pub elf: ElfFile<'static>,  
7  }  
8  
9  pub type AppList = ArrayVec<App, APP_LIST_MAX>;  
10 pub type AppListRef = Option<&'static AppList>;
```

### pkg/boot/src/fs.rs

读取并加载用户程序文件。

```
1  /// Load apps into memory, when no fs implemented in kernel  
2  ///  
3  /// List all file under "APP" and load them.  
4  pub fn load_apps() -> AppList {  
5      let mut root = open_root();  
6      let mut buf = [0; 8];  
7      let cstr_path = uefi::CStr16::from_str_with_buf("\\APP\\", &mut  
8          buf).unwrap();  
9  
10     let mut handle = {  
11         /* DONE: get handle for \APP\ dir */  
12         root.open(cstr_path, FileMode::Read, FileAttribute::empty())  
13             .expect("Failed to open file")  
14             .into_directory()  
15             .expect("Failed to open APP directory")  
16     };
```

```
16
17     let mut apps = ArrayVec::new();
18     let mut entry_buf = [0u8; 0x100];
19
20     loop {
21         let info = handle
22             .read_entry(&mut entry_buf)
23             .expect("Failed to read entry");
24
25         match info {
26             Some(entry) => {
27                 let file = {
28                     /* DONE: get handle for app binary file */
29                     handle.open(entry.file_name(), FileMode::Read,
30                                 FileAttribute::empty())
31                         .expect("Failed to open file");
32
33                     if file.is_directory().unwrap_or(true) {
34                         continue;
35                     }
36
37                     let elf = {
38                         // DONE: load file with `load_file` function
39                         // DONE: convert file to `ElfFile`
40                         let mut reg_file = match
41                             file.into_type().expect("Failed to into_type") {
42                             FileType::Regular(f) => f,
43                             _ => panic!("Invalid file type"),
44                         };
45                         ElfFile::new(load_file(&mut
46                                                 reg_file)).expect("Failed to parse ELF file");
47
48                         let mut name = ArrayString::<16>::new();
49                         entry.file_name().as_str_in_buf(&mut name).unwrap();
50
51                         apps.push(App { name, elf });
52                     }
53                 }
54                 None => break,
```

```
53     }
54 }
55
56 info!("Loaded {} apps", apps.len());
57
58 apps
59 }
```

## 1.2 生成用户程序

### kernel/proc/mod.rs

将 ELF 文件从列表中取出，并生成用户程序。

```
1  pub fn spawn(name: &str) -> Result<ProcessId, String> { Rust
2      let app =
3          x86_64::instructions::interrupts::without_interrupts(|| {
4              let app_list = get_process_manager().app_list()?;
5              app_list.iter().find(|&app| app.name.eq(name))
6          });
7
8      if app.is_none() {
9          return Err(format!("App not found: {}", name));
10     };
11
12     elf_spawn(name.to_string(), &app.unwrap().elf)
13 }
14
15 pub fn elf_spawn(name: String, elf: &ElfFile) -> Result<ProcessId,
16 String> {
17     let pid =
18         x86_64::instructions::interrupts::without_interrupts(|| {
19             let manager = get_process_manager();
20             let process_name = name.to_lowercase();
21
22             let parent = Arc::downgrade(&manager.current());
23
24             let pid = manager.spawn(elf, name, Some(parent), None);
25         });
26 }
```

```

24     debug!("Spawned process: {}#{}", process_name, pid);
25     pid
26 });
27
28 Ok(pid)
29 }

```

### kernel/proc/manager.rs

在 ProcessManager 中实现 spawn 函数。

```

1  pub fn spawn(
2      &self,
3      elf: &ElfFile,
4      name: String,
5      parent: Option<Weak<Process>>,
6      proc_data: Option<ProcessData>,
7  ) -> ProcessId {
8      let kproc = self.get_proc(&KERNEL_PID).unwrap();
9      let page_table = kproc.read().clone_page_table();
10     let proc_vm = Some(ProcessVm::new(page_table));
11     let proc = Process::new(name, parent, proc_vm, proc_data);
12
13     let mut inner = proc.write();
14     inner.pause();
15     inner.load_elf(elf);
16     inner.init_stack_frame(
17         VirtAddr::new_truncate(elf.header.pt2.entry_point()),
18         VirtAddr::new_truncate(super::stack::STACK_INIT_TOP),
19     );
20     drop(inner);
21
22     trace!("New {}:#{?}", &proc);
23
24     let pid = proc.pid();
25     self.add_proc(pid, proc);
26     self.push_ready(pid);
27
28     pid
29 }

```



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

```
1 pub fn load_elf(&mut self, elf: &ElfFile) { Rust
2     let mapper = &mut self.page_table.mapper();
3
4     let alloc = &mut *get_frame_alloc_for_sure();
5
6     self.load_elf_code(elf, mapper, alloc);
7     self.stack.init(mapper, alloc);
8 }
```



### kernel/src/memory/gdt.rs

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。

```
1 lazy_static! { Rust
2     static ref GDT: (GlobalDescriptorTable, KernelSelectors,
3         UserSelectors) = {
4         let mut gdt = GlobalDescriptorTable::new();
5         let code_selector =
6             gdt.append(Descriptor::kernel_code_segment());
7         let data_selector =
8             gdt.append(Descriptor::kernel_data_segment());
9         let tss_selector =
10            gdt.append(Descriptor::tss_segment(&TSS));
11         let user_code_selector =
12            gdt.append(Descriptor::user_code_segment());
13         let user_data_selector =
14            gdt.append(Descriptor::user_data_segment());
15         (
16             gdt,
17             KernelSelectors {
18                 code_selector,
19                 data_selector,
20                 tss_selector,
21             },
22             UserSelectors {
23                 user_code_selector,
```

```

18         user_data_selector,
19     },
20 )
21 };
22 }

```

### 1.3 系统调用的实现

#### kernel/interrupt/syscall/mod.rs

补全系统调用中断注册函数。

```

1  pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) { Rust
2      // FIXME: register syscall handler to IDT
3      //      - standalone syscall stack
4      //      - ring 3
5      unsafe {
6          idt[consts::Interrupts::Syscall as u8]
7              .set_handler_fn(syscall_handler)
8              .set_stack_index(gdt::SYSCALL_IST_INDEX)
9              .set_privilege_level(x86_64::PrivilegeLevel::Ring3);
10     }
11 }

```

#### kernel/interrupt/syscall/service.rs

部份系统调用函数的实现。

```

1  pub fn spawn_process(args: &SyscallArgs) -> usize { Rust
2      // FIXME: get app name by args
3      //      - core::str::from_utf8_unchecked
4      //      - core::slice::from_raw_parts
5      // FIXME: spawn the process by name
6      // FIXME: handle spawn error, return 0 if failed
7      // FIXME: return pid as usize
8      let name = unsafe {
9          core::str::from_utf8_unchecked(core::slice::from_raw_parts(
10              args.arg0 as *const u8,
11              args.arg1,

```



```
12     ))
13 };
14
15     let pid = crate::proc::spawn(name);
16
17     if pid.is_err() {
18         warn!("spawn_process: failed to spawn process: {}", name);
19         return 0;
20     }
21
22     pid.unwrap().0 as usize
23 }
24
25 pub fn sys_write(args: &SyscallArgs) -> usize {
26     // FIXME: get buffer and fd by args
27     //         - core::slice::from_raw_parts
28     // FIXME: call proc::write -> isize
29     // FIXME: return the result as usize
30     let buf = match as_user_slice(args.arg1, args.arg2) {
31         Some(buf) => buf,
32         None => return usize::MAX,
33     };
34
35     let fd = args.arg0 as u8;
36     write(fd, buf) as usize
37 }
38
39 pub fn sys_read(args: &SyscallArgs) -> usize {
40     // FIXME: just like sys_write
41     let buf = match as_user_slice_mut(args.arg1, args.arg2) {
42         Some(buf) => buf,
43         None => return usize::MAX,
44     };
45
46     let fd = args.arg0 as u8;
47     read(fd, buf) as usize
48 }
49
50 pub fn exit_process(args: &SyscallArgs, context: &mut
    ProcessContext) {
```

```

51 // FIXME: exit process with retcode
52 process_exit(args.arg0 as isize, context);
53 }
54
55 pub fn list_process() {
56 // FIXME: list all processes
57 print_process_list();
58 }

```

## 1.4 用户态库的实现

pkg/kernel/src/memory/user.rs

用户态堆初始化。

```

1 pub fn init_user_heap() -> Result<(), MapToError<Size4KiB>>
2 {
3 // Get current pagetable mapper
4 let mapper = &mut PageTableContext::new().mapper();
5 // Get global frame allocator
6 let frame_allocator = &mut *super::get_frame_alloc_for_sure();
7
8 // FIXME: use elf::map_range to allocate & map
9 // frames (R/W/User Access)
10 let page_range = {
11 let heap_start = VirtAddr::new(USER_HEAP_START as u64);
12 let heap_start_page = Page::containing_address(heap_start);
13 let heap_end_page = heap_start_page + USER_HEAP_PAGE as u64
14 - 1u64;
15 Page::range(heap_start_page, heap_end_page)
16 };
17
18 debug!(
19 "User Heap : 0x{:016x}-0x{:016x}",
20 page_range.start.start_address().as_u64(),
21 page_range.end.start_address().as_u64()
22 );
23
24 let (size, unit) = crate::humanized_size(USER_HEAP_SIZE as u64);
25 info!("User Heap Size : {:>7.*} {}", 3, size, unit);

```

```

24
25     for page in page_range {
26         let frame = frame_allocator
27             .allocate_frame()
28             .ok_or(MapToError::FrameAllocationFailed)?;
29         let flags =
30             PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
31             PageTableFlags::USER_ACCESSIBLE;
32         unsafe { mapper.map_to(page, frame, flags,
33             frame_allocator)?.flush() };
34     }
35
36     unsafe {
37         USER_ALLOCATOR
38         .lock()
39         .init(USER_HEAP_START as *mut u8, USER_HEAP_SIZE);
40     }
41
42     Ok(())
43 }

```

### pkg/lib/src/io.rs

标准输入输出的实现。

```

1  pub fn read_line(&self) -> String { Rust
2      // FIXME: allocate string
3      // FIXME: read from input buffer
4      // - maybe char by char?
5      // FIXME: handle backspace / enter...
6      // FIXME: return string
7      let mut string = String::new();
8      loop {
9          let ch = self.pop_key();
10
11          match ch {
12              0x0d => {
13                  stdout().write("\n");
14                  break;
15              }

```

```

16         0x03 => {
17             string.clear();
18             break;
19         }
20         0x08 | 0x7F if !string.is_empty() => {
21             if !string.is_empty() {
22                 stdout().write("\x08 \x08");
23                 string.pop();
24             }
25         }
26         _ => {
27             if Self::is_utf8(ch) {
28                 let utf_char =
29                     char::from_u32(self.to_utf8(ch)).unwrap();
30                 string.push(utf_char);
31                 print! {"{ }", utf_char};
32             } else {
33                 string.push(ch as char);
34                 print! {"{ }", ch as char};
35             }
36         }
37     }
38     string
39 }

```

## 1.5 进程退出

### kernel/proc/mod.rs

进程退出的实现。

```

1 pub fn process_exit(ret: isize, context: &mut ProcessContext)
2 {
3     x86_64::instructions::interrupts::without_interrupts(|| {
4         let manager = get_process_manager();
5         //DONE: implement process exit
6         manager.kill_self(ret);
7         manager.switch_next(context);
8     })
9 }

```



## 1.6 进程创建与等待

### kernel/proc/mod.rs

检查进程是否退出，返回一个 `Option<isize>`。

```
1 pub(crate) fn fetch_exit_code(pid: ProcessId) ->
2   Option<isize> {
3     x86_64::instructions::interrupts::without_interrupts(|| {
4       get_process_manager().get_exit_code(pid)
5     })
6 }
```

### kernel/src/lib.rs

等待进程退出函数 `wait` 的实现。

```
1 pub fn wait(init: proc::ProcessId) {
2   loop {
3     if proc::fetch_exit_code(init).is_none() {
4       x86_64::instructions::hlt();
5     } else {
6       break;
7     }
8   }
9 }
```

## 1.7 Shell 的简单实现

### pkg/app/sh/src/main.rs

Shell 的实现。

```
1 fn main() -> isize {
2   utils::show_welcome_text();
3   loop {
4     utils::print_prompt();
5     let input = stdin().read_line();
6     let line: Vec<&str> = input.trim().split(' ').collect();
7     match line[0] {
8       "\x04" | "exit" => {
```

```
9         println!();
10        break;
11    }
12    "ps" => sys_stat(),
13    "ls" => sys_list_app(),
14    "exec" => {
15        if line.len() < 2 {
16            println!("Usage: exec <file>");
17            continue;
18        }
19
20        services::exec(line[1]);
21    }
22    "kill" => {
23        if line.len() < 2 {
24            println!("Usage: kill <pid>");
25            continue;
26        }
27        let pid = line[1].to_string().parse::<u16>();
28
29        if pid.is_err() {
30            errln!("Cannot parse pid");
31            continue;
32        }
33
34        services::kill(pid.unwrap());
35    }
36    "help" => utils::show_help_text(),
37    "clear" => print!("\x1b[1;1H\x1b[2J"),
38    _ => {
39        if line[0].is_empty() {
40            println!();
41            continue;
42        }
43        println!("Command not found: {}", line[0]);
44        println!("Type 'help' to see available commands.");
45    }
46    }
47 }
48
```

```
49      0  
50 }
```

## 2 思考题

### 2.1 q1

#### ? Question

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。

在现代操作系统中，系统调用主要是为用户态进程设计的，它们通过软中断或其他机制切换到内核态后执行特定操作。内核线程本身已经处于内核态，不需要经过系统调用接口来获得内核权限或执行相应的操作。

因此一般来说，内核线程不会也不应调用系统调用接口（例如 `sys_exit`）来退出进程。对于内核线程来说，更合适的做法是直接调用内核内部的退出或销毁函数，由内核调度器负责清理资源和状态。

如果出于代码统一或抽象考虑希望内核线程也能像用户进程那样“退出”，可以考虑在内核中提供一套接口（例如 `internal_exit()`），该接口内部调用相应的内核方法来终止当前线程（或进程），而不是走系统调用的完整封装路径。

- 系统调用设计用于从用户态进入内核态，而内核线程已经处于内核态，因此没有必要调用系统调用接口。
- 为内核线程实现退出功能，应该直接调用内核的线程/进程退出或清理接口，而非通过 `sys_exit()`。
- 如果需要统一接口，可以设计一份内核内部的“退出”函数，并在不同上下文中分别调用合适的版本。

从设计角度来说，不建议在内核线程中使用系统调用来实现进程退出能力，应直接使用内核内部的接口来完成这一操作。

### 2.2 q2

### ? Question

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。

#### 1. 克隆内核页表的原因

- 为了在每个用户进程中保持一致的内核映射，内核页表（或内核映射部分）通常在各进程页表中共享（或复制）。这样，进程在执行系统调用时进入内核态时，不必切换页表，依然可以访问内核代码和数据。
- 内核映射采用只读或内核特权的标志，这意味着虽然内核页表在所有用户进程中可以看到，但用户态无法直接写入或改变它们，从而保证隔离。

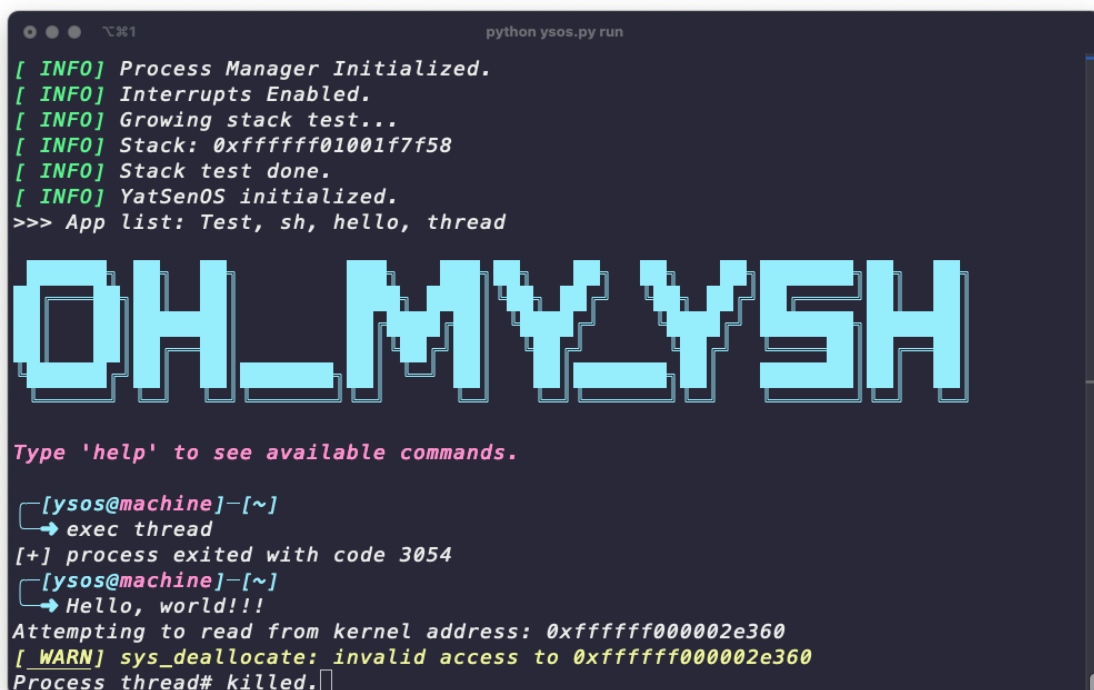
#### 2. 内核态下的页表使用

- 当执行系统调用时，CPU 并不会切换页表，而是直接由当前进程的页表（包含了用户部分和共享的内核部分）提供地址转换。
- 系统调用进入内核态后，仍然使用当前进程的页表，其中内核部分是经过克隆或共享的。

#### 3. 用户态程序访问内核空间

- 内核映射的页表项设置了 `USER_ACCESSIBLE` 标志为 `false`，仅允许在内核态访问。如果用户尝试访问这些地址，会产生页面缺失或保护错误（例如一般保护异常），从而被正确拦截。

#### 4. 验证 访问内核空间地址时，会触发警告（无法访问），如下所示。



```
python ysos.py run
[ INFO] Process Manager Initialized.
[ INFO] Interrupts Enabled.
[ INFO] Growing stack test...
[ INFO] Stack: 0xffffffff01001f7f58
[ INFO] Stack test done.
[ INFO] YatSenOS initialized.
>>> App list: Test, sh, hello, thread

OH_MY_YSH

Type 'help' to see available commands.

[ysos@machine]~
→ exec thread
[+] process exited with code 3054
[ysos@machine]~
→ Hello, world!!!
Attempting to read from kernel address: 0xffffffff000002e360
[ WARN] sys_deallocate: invalid access to 0xffffffff000002e360
Process thread# killed.
```



总的来说，克隆内核页表（或共享其映射部分）使得所有用户进程拥有一致的内核视图，从而在系统调用时无需切换页表；系统调用时实际使用的是当前进程的页表（包括用户与内核部分），而用户态访问内核空间时会因权限不足而触发异常。通过编写实验代码验证对内核地址的非法访问是否被捕获，可以确认这一实现是否正确。

## 2.3 q3

### ? Question

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？

在检查进程状态时关闭中断可以确保操作的原子性，避免在检查执行期间被其他中断或调度操作打断，从而避免数据竞争和状态不一致的问题。具体来说：

- 避免并发修改 如果不关闭中断，可能在读取进程状态时，调度器或其他中断服务程序修改了该状态，比如进程正处在退出过程或状态更新中。这样的话，`still_alive` 函数得到的结果可能就不准确，导致错误的判断。
- 确保原子性 关闭中断等于进入了一个原子临界区，在这个期间可以确保进程的相关数据不会被同时修改，从而保证检查状态的操作是原子的。

总之，如果不关闭中断，可能会因中断导致并发修改而使得进程是否存活的判断出现竞争条件和数据不一致的问题，这会影响整个调度和资源管理的正确性。

## 2.4 q4

### ? Question

对于如下程序，使用 `gcc` 直接编译：

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, World!\n");
5     return 0;
6 }
```

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。

### 1. 编译与链接

- 当你使用 gcc 编译如下代码时，编译器会将 C 源代码转换为汇编，然后交由汇编器和链接器处理。
- 链接器将程序与 C 标准库（比如 libc）以及其他运行时所需的库链接起来，生成一个 ELF 格式的可执行文件（例如默认生成 a.out）。

### 2. 进程创建与加载

- 当你在 shell 中执行该程序时，shell 调用诸如 fork 来创建一个新进程，然后调用 execve 系统调用将可执行文件加载到该进程的地址空间。
- 内核加载器会读取 ELF 文件，将代码段、数据段等映射到新进程的虚拟内存中，同时加载动态链接器（如果需要动态链接库）。

### 3. 执行过程

- 程序通过运行时初始化完成之后会调用程序入口（例如 \_start），随后调用动态链接器，最后调用 main 函数。
- 在 main 中，printf 被调用。此时，printf 会格式化字符串，然后调用底层的系统调用（通常是 write）将格式化后的字符串输出到标准输出（通常是终端设备）。
- 系统调用过程涉及从用户态切换到内核态，经由软中断或特定指令（如 syscall）进入内核中的相应处理函数，完成 I/O 操作后再返回用户态。

### 4. 程序退出

- 当 main 返回 0 时，运行时调用 exit，通知内核该进程退出。
- 内核在接收到退出请求后，会清理进程占用的资源（例如释放内存、关闭文件描述符），并通知父进程（如果父进程调用等待函数）退出状态。

## 2.5 q5

### ? Question

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？

x86\_64::instructions::hlt 指令实际上执行了 CPU 的 HALT 指令，其作用是停止执行指令，进入低功耗（空闲）状态，直到遇到外部中断唤醒 CPU。在内核中，通常在等待过程中调用 hlt，以便在没有任务可运行时让 CPU 进入休眠状态，从而避免忙等待导致的资源浪费。

在用户态中，普通程序是没有权限执行 hlt 指令的，该指令是特权指令。若用户态进程尝试直接调用 hlt，将会触发保护异常（General Protection Fault）。因此，在用

用户态内如 `wait_pid` 之类的等待实现中，不能使用 `hlt`；用户进程应该通过系统调用让内核调度来等待，而内核内部可以使用 `hlt` 来有效降低 CPU 占用。

## 2.6 q6

### ? Question

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……？

x86\_64 架构中，为了确保当 CPU 从用户态（或低特权级别）切换到内核态处理中断/异常时使用的是可靠的栈空间，引入了特权级栈（通常通过 TSS 的中断堆栈表 IST 来指定）。这在以下几个方面起到了关键作用：

1. **中断栈切换** 当发生诸如缺页异常（`0x0e`）这样的异常时，CPU 会自动检查当前特权级与目标特权级是否不同。如果不同（例如从 `Ring3` 切换到 `Ring0`），就会从 TSS 中对应的 IST 索引加载一个专门的内核栈。这样即使原来的用户栈异常或损坏，也能保证中断处理在一个干净、预先分配的环境下进行。
2. **保护临界上下文** 这种栈切换机制避免了在用户栈（可能处于不稳定状态或受攻击的风险中）上执行内核代码，确保内核中断处理期间不会受到意外数据或堆栈溢出问题的干扰。

## 3 实验结果展示

部份 Shell 命令的执行结果如下：



```
python ysos.py run

OH_MY_VSH

Type 'help' to see available commands.

[ysos@machine]~]
→ ps
  PID | PPID | Process Name | Ticks | Status
#  1  | #   0 | kernel       |   114 | Ready
#  2  | #   1 | sh           |   113 | Running
Queue : [1]
CPUs   : [0: 2]
[ysos@machine]~]
→ ls
>>> App list: Test, sh, hello, thread
[ysos@machine]~]
→ exec hello
Hello, world!!!
Process hello# killed.[+] process exited with code 3054
[ysos@machine]~]
→
```