



课程实验报告

0x02 中断处理

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 3 月 6 日

目录

1 GDT 与 TSS	3
2 注册中断处理程序	3
3 初始化 APIC	5
4 时钟中断	7
5 串口输入中断	8
6 思考题	14
6.1 q1	14
6.2 q2	14
6.3 q3	15
6.4 q4	16
6.5 q5	16
6.5.1 triple fault 的触发	17
6.5.2 double_fault 的触发	19
6.5.3 page_fault 的触发	21
6.6 q6	22
6.7 q7	23
7 附加题	23
7.1 q1	24
7.2 q2	27
7.3 q3	28
7.4 q4	29

图表

图 4.0.1 时钟中断效果	7
图 7.4.2 串口输入驱动处理中文和 emoji	29

1 GDT 与 TSS

Task 1

补全 TSS 的中断栈表，为 Double Fault 和 Page Fault 准备独立的栈。

代码如下：

```

1  tss.interrupt_stack_table[(DOUBLE_FAULT_IST_INDEX) as usize] = { Rust
2      const STACK_SIZE: usize = IST_SIZES[1];
3      static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
4      let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK));
5      let stack_end = stack_start + STACK_SIZE as u64;
6      info!(
7          "Double Fault Stack: 0x{:016x}-0x{:016x}",
8          stack_start.as_u64(),
9          stack_end.as_u64()
10     );
11     stack_end
12 };
13
14 tss.interrupt_stack_table[(PAGE_FAULT_IST_INDEX) as usize] = {
15     const STACK_SIZE: usize = IST_SIZES[2];
16     static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
17     let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK));
18     let stack_end = stack_start + STACK_SIZE as u64;
19     info!(
20         "Page Fault Stack : 0x{:016x}-0x{:016x}",
21         stack_start.as_u64(),
22         stack_end.as_u64()
23     );
24     stack_end
25 };
```

2 注册中断处理程序

Task 2

- 为 interrupt 模块添加 pub fn init() 函数，将中断系统的初始化工作统一起来；

- 在 `exception.rs` 中为各种 CPU 异常注册中断处理函数。

初始化函数如下：

```
1  /// init interrupts system Rust
2  pub fn init() {
3      IDT.load();
4
5      // FIXME: check and init APIC
6      unsafe {
7          let mut lapic = XApic::new(physical_to_virtual(LAPIC_ADDR));
8          lapic.cpu_init();
9      }
10     // FIXME: enable serial irq with IO APIC (use enable_irq)
11     enable_irq(irq::Serial0 as u8, 0);
12
13     info!("Interrupts Initialized.");
14 }
```

注册处理函数：

```
1  pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) { Rust
2      idt.divide_error.set_handler_fn(divide_error_handler);
3      idt.debug.set_handler_fn(debug_handler);
4      idt.breakpoint.set_handler_fn(breakpoint_handler);
5      idt.general_protection_fault
6          .set_handler_fn(general_protection_fault_handler);
7      idt.double_fault
8          .set_handler_fn(double_fault_handler)
9          .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX);
10     idt.page_fault
11         .set_handler_fn(page_fault_handler)
12         .set_stack_index(gdt::PAGE_FAULT_IST_INDEX);
13 }
```

部份中断处理函数实现如下：

```
1  pub extern "x86-interrupt" fn debug_handler(stack_frame: Rust
2      InterruptStackFrame) {
3      panic!("EXCEPTION: DEBUG\n\n{:#?}", stack_frame);
4  }
```

```

5  pub extern "x86-interrupt" fn breakpoint_handler(stack_frame:
    InterruptStackFrame) {
6      panic!("EXCEPTION: BREAKPOINT\n\n{:#?}", stack_frame);
7  }
8
9  pub extern "x86-interrupt" fn general_protection_fault_handler(
10     stack_frame: InterruptStackFrame,
11     error_code: u64,
12 ) {
13     panic!(
14         "EXCEPTION: GENERAL PROTECTION FAULT, ERROR_CODE:
            {:#x}\n\n{:#?}",
15         error_code, stack_frame
16     );
17 }
18
19 pub extern "x86-interrupt" fn divide_error_handler(stack_frame:
    InterruptStackFrame) {
20     panic!("EXCEPTION: DIVIDE ERROR\n\n{:#?}", stack_frame);
21 }

```

3 初始化 APIC

初始化代码如下（具体操作已在注释中注明）：

```

1  fn cpu_init(&mut self) {
2      unsafe {
3          // 设置 Spurious Interrupt Vector Register (SVR)
4          let spiv_val = self.read(LapicRegister::SVR);
5          let mut svr = SvrFlags::from_bits_truncate(spiv_val);
6          svr.insert(SvrFlags::ENABLE);
7          svr.remove(SvrFlags::VECTOR_MASK);
8          svr.insert(SvrFlags::from_bits_truncate(Interrupts::IrqBase as
            u32 + Irq::Spurious as u32));
9          self.write(LapicRegister::SVR, svr.bits());
10
11         // 配置定时器除法器 and 初始计数器
12         self.write(LapicRegister::TDCR, 0b1000); // 除法器设置为 1
13         self.write(LapicRegister::TICR, 0x20000); // 初始计数器
14
15         // 设置 LVT Timer 寄存器

```

 Rust

```
16         let lvt_timer_val = self.read(LapicRegister::LvtTimer);
17         let mut lvt_timer =
            LvtTimerFlags::from_bits_truncate(lvt_timer_val);
18         lvt_timer.remove(LvtTimerFlags::VECTOR_MASK);

19         lvt_timer.insert(LvtTimerFlags::from_bits_truncate(Interrupts::Irq
            as u32 + Irq::Timer as u32));
20         lvt_timer.remove(LvtTimerFlags::MASKED);
21         lvt_timer.insert(LvtTimerFlags::PERIODIC);
22         self.write(LapicRegister::LvtTimer, lvt_timer.bits());
23
24         // 屏蔽 LINT0, LINT1 以及 PCINT
25         self.write(LapicRegister::LvtLInt0, 1 << 16);
26         self.write(LapicRegister::LvtLInt1, 1 << 16);
27         self.write(LapicRegister::LvtPCINT, 1 << 16);
28
29         // 设置 LVT Error 寄存器
30         let lvt_error_val = self.read(LapicRegister::LvtError);
31         let mut lvt_error =
            LvtErrorFlags::from_bits_truncate(lvt_error_val);
32         lvt_error.remove(LvtErrorFlags::VECTOR_MASK);

33         lvt_error.insert(LvtErrorFlags::from_bits_truncate(Interrupts::Irq
            as u32 + Irq::Error as u32));
34         lvt_error.remove(LvtErrorFlags::MASKED);
35         self.write(LapicRegister::LvtError, lvt_error.bits());
36
37         // 清除错误状态寄存器两次以清除旧错误
38         self.write(LapicRegister::ESR, 0);
39         self.write(LapicRegister::ESR, 0);
40
41         // 发出 End-Of-Interrupt
42         self.eoi();
43         // 发送 INIT Level De-assert 命令 (示例代码)
44         const BCAST: u32 = 1 << 19;
45         const INIT: u32 = 5 << 8;
46         const TMLV: u32 = 1 << 15; // TM = 1, LV = 0
47         self.set_icr((BCAST | INIT | TMLV) as u64);
48         // 设置 Task Priority Register (TPR) 为 0 以允许所有中断
49         self.write(LapicRegister::TPR, 0);
50     }
51 }
```

4 时钟中断

Task 3

- 补全上述代码任务，并尝试修改你的代码，调节时钟中断的频率，并观察 QEMU 中的输出。
- 说明你修改了哪些代码，如果想要中断的频率减半，应该如何修改？

可以直接通过修改 TDCR 寄存器的值来调节时钟中断的频率。根据分频系数的不同，时钟中断的频率也会有所变化。比如，如果想要将时钟中断的频率减半，可以将 TDCR 寄存器的值从 0b1000 修改为 0b1001（分频系数从 32 变为了 64）。

```
1 // 配置定时器除法器 and 初始计数器 Rust
2 self.write(LapicRegister::TDCR, 0b1000); // 除法器设置为 by 32
3 self.write(LapicRegister::TICR, 0x20000); // 初始计数器
```

可以观察到时钟中断成功在 QEMU 中触发，并且在日志中输出了 Tick。

```
python ysos.py run

Student ID: 23336003
[+] Serial Initialized.
[ INFO] Logger Initialized. (log_level = info)
[ INFO] Physical Offset : 0xffff800000000000
[ INFO] Privilege Stack : 0xffffffff000081b410-0xffffffff000081c410
[ INFO] Double Fault Stack: 0xffffffff000081c410-0xffffffff000081d410
[ INFO] Page Fault Stack : 0xffffffff000081d410-0xffffffff000081e410
[ INFO] Kernel IST Size : 12.000 KB
[ INFO] GDT Initialized.
[ INFO] Kernel Heap Size : 8.000 MB
[ INFO] Kernel Heap Initialized.
[ INFO] Interrupts Initialized.
[ INFO] Physical Memory : 95.625 MB
[ INFO] Free Usable Memory : 44.984 MB
[ INFO] Frame Allocator initialized.
[ INFO] Interrupts Enabled.
[ INFO] YatSenOS initialized.
[ INFO] Hello World from YatSenOS v2!
[ INFO] Tick! @1
```

图 4.0.1: 时钟中断效果

5 串口输入中断

Task 4

按照下列描述，补全 `src/drivers/input.rs` 驱动代码：

1. 使用你喜欢的数据结构存储用户输入的数据。
2. 处理数据结构的初始化，暴露基本功能。
3. 实现并暴露 `pop_key` 函数。
4. 实现并暴露 `get_line` 函数。
5. 串口的输入中断与时钟中断类似，在 `src/interrupt/serial.rs` 中补全代码，为 **IRQ4 Serial0** 设置中断处理程序。
6. 最后，你需要补全 `receive` 函数，利用刚刚完成的 `input` 驱动，将接收到的字符放入缓冲区。

`src/drivers/input.rs`:

```

1  // 内核环境中提供了类似的输出宏，输出到串口或其他终端设备
2  /// 定义输入数据类型，此处使用 u8 表示单个字符的 ASCII 码
3  pub type Key = u8;
4  lazy_static! {
5      // 初始化一个大小为 128 的无锁队列作为输入缓冲区
6      static ref INPUT_BUF: ArrayQueue<Key> = ArrayQueue::new(128);
7
8      pub static ref HISTORY: Mutex<Vec<String>> = Mutex::new(Vec::new());
9  }
10
11 /// 将一个键值压入输入缓冲区
12 #[inline]
13 pub fn push_key(key: Key) {
14     if INPUT_BUF.push(key).is_err() {
15         // 如果缓冲区已满，输出警告（请确保 warn! 宏已在内核中定义）
16         warn!("Input buffer is full. Dropping key '{:?}'", key);
17     }
18 }
19
20 /// 尝试从输入缓冲区非阻塞地取出一个键值
21 #[inline]
22 pub fn try_pop_key() -> Option<Key> {
23     INPUT_BUF.pop()

```



```
24 }
25
26 /// 阻塞地从输入缓冲区获取一个键值，直到有数据为止
27 pub fn pop_key() -> Key {
28     loop {
29         if let Some(key) = try_pop_key() {
30             return key;
31         }
32         spin_loop();
33     }
34 }
35 pub fn get_line() -> String {
36     let mut input_buffer: Vec<u8> = Vec::with_capacity(128);
37     let prompt = "> ";
38     let mut history_index: Option<usize> = None;
39     // 记录光标在 input_buffer 中的字节索引（必须始终在有效字符边界上）
40     let mut cursor_pos: usize = 0;
41
42     print!("{}", prompt);
43
44     loop {
45         let key = pop_key();
46
47         // 检查是否为转义序列（ESC 开头）
48         if key == 0x1B {
49             let second = pop_key();
50             let third = pop_key();
51             if second == b'[' {
52                 match third {
53                     b'A' => {
54                         // 上箭头：历史向上
55                         let history = HISTORY.lock();
56                         if !history.is_empty() {
57                             history_index = match history_index {
58                                 None => Some(history.len() - 1),
59                                 Some(id) if id > 0 => Some(id - 1),
60                                 Some(id) => Some(id),
61                             };
62                             if let Some(id) = history_index {
63                                 input_buffer.clear();
64
65                                 input_buffer.extend_from_slice(history[id
```

```
65                                     // 历史加载后, 将光标置于末尾
66                                     cursor_pos = input_buffer.len();
67                                     }
68                                 }
69                            }
70                            b'B' => {
71                                // 下箭头: 历史向下
72                                let history = HISTORY.lock();
73                                if !history.is_empty() {
74                                    history_index = match history_index {
75                                        Some(idy) if idy < history.len() - 1 =>
76                                            Some(idy + 1),
77                                        _ => None,
78                                    };
79                                    input_buffer.clear();
80                                    if let Some(idy) = history_index {
81                                        input_buffer.extend_from_slice(history[idy]);
82                                        cursor_pos = input_buffer.len();
83                                    }
84                                }
85                                b'D' => {
86                                    // 左箭头: 向左移动光标
87                                    if cursor_pos > 0 {
88                                        if let Ok(s) =
89                                            core::str::from_utf8(&input_buffer) {
90                                            if let Some((idy, _)) =
91                                                s[..cursor_pos].char_indices().rev().next()
92                                            {
93                                                cursor_pos = idy;
94                                            } else {
95                                                cursor_pos -= 1;
96                                            }
97                                        } else {
98                                            cursor_pos -= 1;
99                                        }
100                                    }
101                                }
102                                b'C' => {
103                                    // 右箭头: 向右移动光标
104                                    if cursor_pos < input_buffer.len() {
```

```
102         if let Ok(s) =
               core::str::from_utf8(&input_buffer) {
103             if let Some((rel_idx, ch)) =
                   s[cursor_pos..].char_indices().next() {
104                 cursor_pos += rel_idx +
                           ch.len_utf8();
105             } else {
106                 cursor_pos = input_buffer.len();
107             }
108         } else {
109             cursor_pos += 1;
110         }
111     }
112 }
113 _ => {
114     // 忽略其他转义序列
115     continue;
116 }
117 }
118 // 每个转义序列处理后，刷新显示并定位光标
119 let current_input =
core::str::from_utf8(&input_buffer).unwrap_or("<invalid
utf8>");
120 print!("\r\x1B[K{}", prompt, current_input);
121 // 安全地获取前 cursor_pos 子串，避免越界
122 let safe_cursor_pos = if cursor_pos >
current_input.len() { current_input.len() } else
{ cursor_pos };
123 let displayed_cursor =
current_input.get(..safe_cursor_pos).unwrap_or("").width(
124 print!("\r\x1B[{}C", prompt.width() + displayed_cursor);
125 continue;
126 }
127 }
128
129 if key == b'\n' || key == b'\r' {
130     // 回车，输入结束。刷新行后退出循环
131     print!("\n");
132     break;
133 } else if key == 0x08 || key == 0x7F {
134     // 退格：删除光标前一个字符（按 UTF-8 分界安全删除）
```

```
135         if cursor_pos > 0 {
136             if let Ok(s) = core::str::from_utf8(&input_buffer) {
137                 if let Some((idx, _)) =
138                     s[..cursor_pos].char_indices().rev().next() {
139                     input_buffer.drain(idx..cursor_pos);
140                     cursor_pos = idx;
141                 } else {
142                     input_buffer.drain((cursor_pos -
143                     1)..cursor_pos);
144                     cursor_pos -= 1;
145                 }
146             } else {
147                 input_buffer.drain((cursor_pos - 1)..cursor_pos);
148                 cursor_pos -= 1;
149             }
150         } else {
151             // 普通字符输入：插入到 cursor_pos 处，并移动光标
152             history_index = None;
153             input_buffer.insert(cursor_pos, key);
154             cursor_pos += 1;
155         }
156
157         let current_input =
158             core::str::from_utf8(&input_buffer).unwrap_or("<invalid utf8>");
159         // 刷新整行显示：\r 回到行首，\x1B[K 清除行尾，然后打印提示符+内容
160         print!("\r\x1B[K{}]", prompt, current_input);
161         // 同样安全地计算光标位置
162         let safe_cursor_pos = if cursor_pos > current_input.len()
163             { current_input.len() } else { cursor_pos };
164         let displayed_cursor =
165             current_input.get(..safe_cursor_pos).unwrap_or("").width();
166         print!("\r\x1B[{}C", prompt.width() + displayed_cursor);
167     }
168
169     let final_command = core::str::from_utf8(&input_buffer)
170         .unwrap_or("<invalid utf8>")
171         .to_owned();
172     if !final_command.is_empty() {
173         HISTORY.lock().push(final_command.clone());
174     }
```

```

172     final_command
173 }

```

串口中断 src/interrupt/serial.rs:

```

1  /// 使用 UART 16550 的基地址 (假设为 COM1, 即 0x3F8) Rust
2  const UART_PORT: u16 = 0x3F8;
3  /// 注册串口中断处理函数
4  pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
5      idt[Interrupts::IrqBase as u8 + Irq::Serial0 as u8]
6          .set_handler_fn(serial_handler);
7  }
8  /// 串口中断处理函数
9  pub extern "x86-interrupt" fn serial_handler(_st: InterruptStackFrame) {
10     receive();
11     super::ack();
12 }
13 /// 从串口 16550 读取数据并放入输入缓冲区
14 fn receive() {
15     unsafe {
16         // 检查 Line Status Register (LSR) (端口 UART_PORT + 5) 中 Data
17         // Ready 位是否为 1
18         if inb(UART_PORT + 5) & 1 != 0 {
19             // 读取接收数据寄存器 (UART_PORT 处)
20             let ch = inb(UART_PORT);
21             push_key(ch);
22         }
23     }
24     /// 从指定端口读取一个字节
25     #[inline]
26     unsafe fn inb(port: u16) -> u8 {
27         let mut value: u8;
28         core::arch::asm!(
29             "in al, dx",
30             out("al") value,
31             in("dx") port,
32             options(nomem, nostack, preserves_flags)
33         );
34         value
35     }

```

6 思考题

6.1 q1

? Question

为什么需要在 `clock_handler` 中使用 `without_interrupts` 函数？如果不使用它，可能会发生什么情况？

在时钟中断处理器中调用 `without_interrupts` 可以暂时禁止中断进入，从而保证在整个处理过程期间不会被中断打断。

- 保护临界区：在 `clock_handler` 中，可能会修改全局状态（例如 COUNTER 计数器）以及执行如打印输出等操作。如果在处理中再次收到中断，就可能导致数据竞争或不一致的状态。

- 防止嵌套中断：如果不禁止中断，可能会发生嵌套中断（例如时钟中断在处理前一次时钟中断时再次触发），可能导致堆栈溢出或 double fault。

6.2 q2

? Question

考虑时钟中断进行进程调度的场景，时钟中断的频率应该如何设置？太快或太慢的频率会带来什么问题？请分别回答。

- 频率过快
 - 会导致中断处理频繁触发，增加系统开销。
 - 上下文切换次数增多，降低任务真正执行的时间，整体性能可能下降。
 - 系统可能消耗大量 CPU 时间处理中断，而非实际工作。
- 频率过慢
 - 进程调度响应变得迟缓，降低系统的交互性与实时性。
 - 任务之间切换间隔过长，可能导致部分任务长时间无法获得处理机会。
 - 对于需要快速响应的场景，比如实时应用，可能会影响用户体验和系统稳定性。

因此，合理的时钟中断频率需要视具体应用场景决定，通常在几十到几百赫兹之间较为合适，以兼顾响应速度和系统开销。

6.3 q3

? Question

在进行 receive 操作的时候，为什么无法进行日志输出？如果强行输出日志，会发生什么情况？谈谈你对串口、互斥锁的认识。

1. 中断处理程序要求快速完成

中断处理程序应尽快执行完毕，避免长时间占用 CPU。如果在中断中调用日志输出，往往会涉及格式化字符串、内存分配以及 I/O 操作，这会大幅增加中断处理时间，从而影响系统实时性。

2. 锁（互斥锁）不可重入问题

日志输出通常需要获取全局锁（例如控制台或串口输出锁）以保证输出内容不互相交织。中断处理程序被调用时可能在其他上下文中已经持有同一把锁（例如在主线程中正在输出日志），这时如果中断又尝试获取该锁，会导致死锁或者自旋等待。

3. 串口驱动和日志输出的关联

在当前设计中，串口既被用作设备输入又用于输出日志。如果在串口的接收中断处理程序中进行日志输出，可能会出现递归调用（即日志输出也需要串口输出，而串口输出又可能产生中断），这种情形常常会导致堆栈溢出或死锁。

如果强行在 **receive** 中调用日志输出：

- 可能会导致系统死锁，因为日志输出内部通常会使用互斥锁保护串口输出，而这把锁可能在中断前后已经被其他任务持有。
- 日志输出中的额外开销会延长中断处理时间，可能引发时钟中断嵌套或其他实时性问题。
- 更严重的情况下，如果日志输出触发了再次中断调用（例如串口中断），可能形成递归调用，使堆栈溢出，从而引发双重故障。

对串口与互斥锁的认识：

• 串口

串口作为一种简单的 I/O 设备，既用于数据传输也常被用作调试输出渠道。其操作通常比较低级，涉及端口 I/O 指令。在内核中操作串口时必须保证操作简洁、高效，而且通常需要在中断处理程序中进行迅速的数据读取或写入。串口的输出函数常常是非阻塞的，并利用锁保证多核或多上下文环境下的互斥访问。

• 互斥锁

互斥锁用于保护共享资源，防止多个执行单元同时访问造成数据竞争。但在内核中（尤其是中断上下文）应避免使用会阻塞或等待的锁，因为中断处理程序要求快速且不可阻塞的执行。常见做法是在中断上下文使用自旋锁（spinlock），但即便如此，也必须小心使用，避免在已经持有锁的代码中再次调用会获取同一把锁的代码，从而导致自旋死锁。

综上，在串口接收中断中调用日志输出会破坏中断上下文的实时性，并可能引发锁竞争和死锁问题，因此应避免这种设计。

6.4 q4

? Question

输入缓冲区在什么情况下会满？如果缓冲区满了，用户输入的数据会发生什么情况？

在代码中，输入缓冲区采用了一个固定大小为 128 的无锁队列（ArrayQueue），最多只能缓存 128 个按键（Key）。当输入的按键超过 128 个而还没有被及时消费（通过 pop_key 读取），队列就会满。

如果缓冲区满了，那么在 push_key 函数中调用 push 操作时会返回错误。该代码对此做了处理，调用 warn! 输出一条警告信息，并且丢弃新来的按键：

```
1 if INPUT_BUF.push(key).is_err() {  
2     warn!("Input buffer is full. Dropping key '{:?}'", key);  
3 }
```

 Rust

即，当缓冲区达到容量上限时，用户输入的额外数据不会被保存并进入队列，而是直接丢弃（提示警告），从而导致用户输入的部分按键失效。

6.5 q5

? Question

进行下列尝试，并在报告中保留对应的触发方式及相关代码片段：

- 尝试用你的方式触发 Triple Fault，开启 intdbg 对应的选项，在 QEMU 中查看调试信息，分析 Triple Fault 的发生过程。
- 尝试触发 Double Fault，观察 Double Fault 的发生过程，尝试通过调试器定位 Double Fault 发生时使用的栈是否符合预期。

- 通过访问非法地址触发 Page Fault，观察 Page Fault 的发生过程。分析 Cr2 寄存器的值，并尝试回答为什么 Page Fault 属于可恢复的异常。

6.5.1 triple fault 的触发

可以在主程序中加入一个无限递归调用的函数，使得在运行时触发 `page_fault`，并取消对 `IST` 的设置。这样又会触发 `double_fault`。最后取消对 `double_fault` 的设置，从而导致 `triple fault`。

触发代码如下（部份）：

无限递归调用

```
1 // 声明一个无条件递归函数，用于触发堆栈溢出
2 #[allow(unconditional_recursion)]
3 fn trigger_stack_overflow() {
4     trigger_stack_overflow();
5 }
```

 Rust

取消对 `IST` 的设置

```
1 idt.double_fault
2     .set_handler_fn(double_fault_handler);
3     // .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX);
4 idt.page_fault
5     .set_handler_fn(page_fault_handler);
6     // .set_stack_index(gdt::PAGE_FAULT_IST_INDEX);
```

 Rust

最后在 `qemu` 中运行，观察到 `triple fault` 触发后，系统反复重启。启用 `intdbg` 选项后，可以看到如下输出（部份）：

```
1 Servicing hardware INT=0x68
   67: v=68 e=0000 i=0 cpl=0 IP=0038:0000000005f16337
2   pc=0000000005f16337 SP=0030:0000000005f107a0 env-
   >regs[R_EAX]=8000000000000002
   RAX=8000000000000002 RBX=0000000000000004 RCX=0000000005455e98
3   RDX=0000000004ae8ae0
   RSI=0000000000000000 RDI=000000000000003e RBP=0000000004ae8a60
4   RSP=0000000005f107a0
   R8 =0000000000000000 R9 =0000000004b16c98 R10=0000000005f279a0
5   R11=0000000000000000
   R12=0000000005455e98 R13=00000000044d6698 R14=00000000044d6d01
6   R15=0000000005455e98
```

 md

7	RIP=0000000005f16337	RFL=00000202	[-----]	CPL=0	II=0	A20=1	SMM=0	HLT=0
8	ES =0030	0000000000000000	ffffffff	00cf9300	DPL=0	DS	[-WA]	
9	CS =0038	0000000000000000	ffffffff	00af9a00	DPL=0	CS64	[-R-]	
10	SS =0030	0000000000000000	ffffffff	00cf9300	DPL=0	DS	[-WA]	
11	DS =0030	0000000000000000	ffffffff	00cf9300	DPL=0	DS	[-WA]	
12	FS =0030	0000000000000000	ffffffff	00cf9300	DPL=0	DS	[-WA]	
13	GS =0030	0000000000000000	ffffffff	00cf9300	DPL=0	DS	[-WA]	
14	LDT=0000	0000000000000000	0000ffff	00008200	DPL=0	LDT		
15	TR =0000	0000000000000000	0000ffff	00008b00	DPL=0	TSS64-busy		
16	GDT=	00000000059de000	00000047					
17	IDT=	0000000005472018	00000fff					
18	CR0=80010033	CR2=0000000000000000	CR3=0000000005c01000	CR4=00000668				
19	DR0=0000000000000000	DR1=0000000000000000	DR2=0000000000000000	DR3=0000000000000000				
20	DR6=00000000ffff0fff	DR7=00000000000000400						
21	CCS=0000000000000000	CCD=0000000004ae8ae0	CC0=EFLAGS					
22	EFER=00000000000000d00							

根据 QEMU 调试日志中 “Servicing hardware INT=0x68” 输出，分析 triple fault 触发过程：

1. 连续异常和异常链中断 日志连续多次显示对中断向量 0x68 的处理。这表明在某一时刻，一个异常刚刚触发后，处理该异常的过程中又发生了新的异常。按照 x86 架构的规则：
 - 第一次异常（例如由无限递归或堆栈溢出引起的 page fault）被触发后，CPU 会尝试调用该异常的处理程序。
 - 但如果在异常处理过程中（例如由于堆栈损坏或资源耗尽导致）再次发生错误，就会触发 double fault。
 - 而如果 double fault 处理程序自身由于配置问题无法正确执行，则会导致 triple fault，最终使 CPU 进入复位。
2. 从寄存器状态中的异常迹象看 查看日志中的寄存器转储信息，我们可以注意到：
 - 寄存器信息显示了一个正常进入异常处理的现场；例如 RSP（栈指针）和 RIP（指令指针）正处于异常处理入口附近。
 - 随后，寄存器状态出现了明显异常：如 RAX 的值变为 0x8000000000000002，这通常表示某个异常状态标志已经被设置，同时其他寄存器（如 RDI、RBP 等）的值也发生了异常变化。这种状态混乱表明异常处理链已经被破坏，无法正常恢复。
3. 异常链条失效—Triple Fault 的形成 根据以上现象，我们可以推测出：
 - 系统首先因堆栈溢出进入第一级异常（page_fault）；

- 在处理这个异常时，由于堆栈环境已损坏，导致异常处理过程中又发生了第二个异常（即 double fault）；
- 由于 double fault handler 被刻意配置为“无效”，使得 double fault 处理也失败；
- 当连续两次异常均无法被正确处理时，CPU 根据 x86 规则触发 triple fault，最终导致系统无条件复位（在 QEMU 下，配合 `-no-reboot` 参数，可以看到复位之前最后的调试日志）。

4. 从日志中识别

- 看到连续的 “Servicing hardware INT=0x68” 日志记录表明 CPU 正在不断中断服务过程中异常地进入异常处理层级。
- 每一次记录的寄存器状态都向我们展示了处理现场：当异常处理链已失效时，寄存器（例如 RAX 出现异常大数值、RSP 不连续变化等）显示出不合理的信息。
- 最终，QEMU 输出的调试信息中（系统停顿），就证明了异常处理已进入无可恢复的状态。

综上，根据这些调试日志，可以判断：初始异常触发后，由于堆栈环境被耗尽或破坏，异常处理过程中再次出错，导致 double fault handler 被调用；而我们刻意让 double fault handler 无效，使得第二次异常继续失败，进而触发 triple fault。只要观察到寄存器状态出现异常、不连续变化，以及 QEMU 在 `-no-reboot` 模式下停下不复位的信息，就可以确认 triple fault 的触发过程。

6.5.2 double_fault 的触发

取消为 Page Fault 分配 IST，从而让异常处理时使用当前受损的栈；同时配置一个简单的 Double Fault handler（例如通过 `panic!` 或直接停止）以便观察现场。


```
1 idt.double_fault
2     .set_handler_fn(double_fault_handler)
3     .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX);
4 idt.page_fault
5     .set_handler_fn(page_fault_handler);
6     // .set_stack_index(gdt::PAGE_FAULT_IST_INDEX);
```

 Rust

与上述 triple fault 类似，double fault 触发的原因是：在处理 page fault 时，栈已经损坏或不可用，导致无法正确返回到异常处理程序。此时，CPU 会尝试调用 double fault handler，从而导致 double fault。同样使用无限递归的方式触发。

此时，会得到错误信息：

```
1 [ERROR] ERROR: panic!
```

 Text

```

2
3 PanicInfo {
4     message: EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x0000000000000000
5
6     InterruptStackFrame {
7         instruction_pointer: VirtAddr(
8             0xffffffff000000c1a0,
9         ),
10        code_segment: SegmentSelector {
11            index: 1,
12            rpl: Ring0,
13        },
14        cpu_flags: RFlags(
15            INTERRUPT_FLAG | SIGN_FLAG | AUXILIARY_CARRY_FLAG |
16            PARITY_FLAG | 0x2,
17        ),
18        stack_pointer: VirtAddr(
19            0xffffffff0100000000,
20        ),
21        stack_segment: SegmentSelector {
22            index: 0,
23            rpl: Ring0,
24        },
25        location: Location {
26            file: "pkg/kernel/src/interrupt/exceptions.rs",
27            line: 50,
28            col: 5,
29        },
30        can_unwind: true,
31        force_no_backtrace: false,
32    }

```

同时，我们通过 lldb 调试器可以看到寄存器状态：

```

rsp = 0xffffffff0000867df8 KERNEL.ELF`<ysos_kernel::memory::gdt::TSS
1 as core::ops::deref::Deref>::deref::__static_ref_initialize::STACK Rust
+ 3504

```

根据先前为 double fault 分配的 IST 栈地址 0xffffffff0000867048-0xffffffff0000868048，可以看到 rsp 寄存器指向了 double fault 的栈空间。

6.5.3 page_fault 的触发

我们恢复对 Page Fault IST 的设置。依然通过无限递归调用，来触发 page fault。

```

1 idt.double_fault
2     .set_handler_fn(double_fault_handler)
3     .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX);
4 idt.page_fault
5     .set_handler_fn(page_fault_handler)
6     .set_stack_index(gdt::PAGE_FAULT_IST_INDEX);

```

可以得到如下错误信息：

```

1 [ERROR] ERROR: panic!
2
3 PanicInfo {
4     message: EXCEPTION: PAGE FAULT, ERROR_CODE:
      PageFaultErrorCode(CAUSED_BY_WRITE)
5
6     Trying to access: 0xffffffff00000008
7     InterruptStackFrame {
8         instruction_pointer: VirtAddr(
9             0xffffffff000000c1a0,
10        ),
11        code_segment: SegmentSelector {
12            index: 1,
13            rpl: Ring0,
14        },
15        cpu_flags: RFlags(
16            RESUME_FLAG | INTERRUPT_FLAG | SIGN_FLAG |
17            AUXILIARY_CARRY_FLAG | PARITY_FLAG | 0x2,
18        ),
19        stack_pointer: VirtAddr(
20            0xffffffff0100000000,
21        ),
22        stack_segment: SegmentSelector {
23            index: 0,
24            rpl: Ring0,
25        },
26        location: Location {
27            file: "pkg/kernel/src/interrupt/exceptions.rs",

```

```
28         line: 60,  
29         col: 5,  
30     },  
31     can_unwind: true,  
32     force_no_backtrace: false,  
33 }
```

通过 lldb 调试器可以看到寄存器状态：

```
1 cr2: 0xffffffff00000000
```

 Rust

Page Fault 是一种可恢复的异常，处理过程大致如下：

1. 触发条件和 Cr2 寄存器 当 CPU 试图访问一个无效或未映射的内存地址时，会触发 Page Fault 异常。此时，硬件会自动将导致异常的线性地址写入 Cr2 寄存器。在我们的 `page_fault_handler` 中，通过调用 `Cr2::read()` 得到该地址，例如输出信息：

```
1 "Trying to access: {:#x}"
```

这说明 Cr2 存储了引发页面错误的地址。

2. 异常处理机制与可恢复性 Page Fault 被归类为可恢复异常，因为：

- 异常处理程序已注册：操作系统在 IDT 中为 Page Fault 注册了处理程序 (`page_fault_handler`)，处理时可以检查错误码以及 Cr2 中存储的地址。
- 可能的恢复策略：在实际操作系统中，基于错误码，页面错误处理程序可以判断是缺页 (`page not present`)，还是写保护冲突等，并通过分配新页、加载页面或杀死产生异常的进程等方式恢复执行。
- 异常返回机制：成功处理 Page Fault 后，处理程序通常会调用一条返回指令（例如 `iretq`），继续执行引发异常的指令，从而实现“恢复”操作。

总结来说，Page Fault 属于可恢复的异常，因为在发生此异常时，CPU 会保存引起错误的地址（在 Cr2 中），并调用操作系统注册的异常处理程序；处理程序可以根据错误码采取适当措施恢复执行（例如加载缺失的页面），再通过返回指令使程序继续运行。这与不可恢复的异常（如 Double Fault 或 Triple Fault）不同，后者往往没有有效的恢复路径，通常会导致系统崩溃或重启。

6.6 q6

? Question

如果在 TSS 中为中断分配的栈空间不足，会发生什么情况？请分析 CPU 异常的发生过程，并尝试回答什么时候会发生 Triple Fault。

1. 处理异常时，CPU 会根据 IDT 中设置的 IST（中断栈表）从 TSS 选择一个专用栈来处理异常。如果该专用栈区域设置得太小或不够用，异常处理程序在开始执行时就可能在栈空间不足处发生栈溢出。
2. 假设发生 Page Fault 时，处理程序切换到预设的 IST 栈上，但栈空间不足，执行过程中又产生新的异常（例如缺乏足够空间写入现场或保存寄存器值），这时原本的异常处理程序无法正确运行。CPU 检测到当前异常处理期间再次出错，就会尝试调用 Double Fault handler。
3. 如果 Double Fault handler 同样因为使用的栈（也可能是同一块不足的 IST）不足而无法正常运行，也就是在处理 Double Fault 时再次发生错误，那么根据 x86 架构的规则，CPU 就会检测到 Triple Fault。
4. 当 Triple Fault 发生时，CPU 无法从异常中恢复，最终会导致系统复位（在 QEMU 中通常表现为“Triple fault”后系统停止或重置）。

6.7 q7

? Question

在未使用 `set_stack_index` 函数时，中断处理程序的栈可能哪里？尝试结合 gdb 调试器，找到中断处理程序的栈，并验证你的猜想是否正确。

在没有调用 `set_stack_index` 指定专用 IST 栈的情况下，CPU 会默认使用产生异常时的当前栈（通常是任务的内核栈）来处理中断。这意味着，当异常（例如 page fault 或 double fault）发生时，CPU 不会切换到事先配置好的、独立的中断栈，而是使用正在运行的那个栈。如果这个栈已经由于前一次异常而损坏或溢出，那么中断处理程序就可能在一个不可靠的栈上运行，从而加剧问题。

在 q5 中，我们已经看到，如果在 page fault 或 double fault 处理程序中没有设置 IST 栈，可能会导致 double fault 或 triple fault。

7 附加题

7.1 q1

? Question

😄 为全部可能的 CPU 异常设置对应的处理程序，使用 `panic!` 输出异常信息。

代码如下：

```

1  pub extern "x86-interrupt" fn divide_error_handler(stack_frame:
    InterruptStackFrame) {
2      panic!("EXCEPTION: DIVIDE ERROR\n{:#?}", stack_frame);
3  }
4
5  pub extern "x86-interrupt" fn debug_handler(stack_frame:
    InterruptStackFrame) {
6      panic!("EXCEPTION: DEBUG\n{:#?}", stack_frame);
7  }
8
9  // 非屏蔽中断 (NMI)
10 pub extern "x86-interrupt" fn nmi_handler(stack_frame:
    InterruptStackFrame) {
11     panic!("EXCEPTION: NON-MASKABLE INTERRUPT\n{:#?}", stack_frame);
12 }
13
14 pub extern "x86-interrupt" fn breakpoint_handler(stack_frame:
    InterruptStackFrame) {
15     panic!("EXCEPTION: BREAKPOINT\n{:#?}", stack_frame);
16 }
17
18 pub extern "x86-interrupt" fn overflow_handler(stack_frame:
    InterruptStackFrame) {
19     panic!("EXCEPTION: OVERFLOW\n{:#?}", stack_frame);
20 }
21
22 pub extern "x86-interrupt" fn bound_range_exceeded_handler(
23     stack_frame: InterruptStackFrame,
24 ) {
25     panic!("EXCEPTION: BOUND RANGE EXCEEDED\n{:#?}", stack_frame);
26 }
27

```



```
28 pub extern "x86-interrupt" fn invalid_opcode_handler(stack_frame:
    InterruptStackFrame) {
29     panic!("EXCEPTION: INVALID OPCODE\n{:#?}", stack_frame);
30 }
31
32 pub extern "x86-interrupt" fn device_not_available_handler(
33     stack_frame: InterruptStackFrame,
34 ) {
35     panic!("EXCEPTION: DEVICE NOT AVAILABLE\n{:#?}", stack_frame);
36 }
37
38 pub extern "x86-interrupt" fn double_fault_handler(
39     stack_frame: InterruptStackFrame,
40     error_code: u64,
41 ) -> ! {
42     panic!(
43         "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x{:016x}\n{:#?}",
44         error_code, stack_frame
45     );
46 }
47
48 pub extern "x86-interrupt" fn invalid_tss_handler(
49     stack_frame: InterruptStackFrame,
50     error_code: u64,
51 ) {
52     panic!(
53         "EXCEPTION: INVALID TSS, ERROR_CODE: 0x{:016x}\n{:#?}",
54         error_code, stack_frame
55     );
56 }
57
58 pub extern "x86-interrupt" fn segment_not_present_handler(
59     stack_frame: InterruptStackFrame,
60     error_code: u64,
61 ) {
62     panic!(
63         "EXCEPTION: SEGMENT NOT PRESENT, ERROR_CODE: 0x{:016x}\n{:#?}",
64         error_code, stack_frame
65     );
66 }
67
68 pub extern "x86-interrupt" fn stack_segment_fault_handler(
```

```
69     stack_frame: InterruptStackFrame,
70     error_code: u64,
71 ) {
72     panic!(
73         "EXCEPTION: STACK SEGMENT FAULT, ERROR_CODE: 0x{:016x}\n{:#?}",
74         error_code, stack_frame
75     );
76 }
77
78 pub extern "x86-interrupt" fn general_protection_fault_handler(
79     stack_frame: InterruptStackFrame,
80     error_code: u64,
81 ) {
82     panic!(
83         "EXCEPTION: GENERAL PROTECTION FAULT, ERROR_CODE:
84         0x{:016x}\n{:#?}",
85         error_code, stack_frame
86     );
87 }
88
89 pub extern "x86-interrupt" fn page_fault_handler(
90     stack_frame: InterruptStackFrame,
91     err_code: PageFaultErrorCode,
92 ) {
93     panic!(
94         "EXCEPTION: PAGE FAULT, ERROR_CODE: {:?}\nAccessed Address:
95         {:#x}\n{:#?}",
96         err_code,
97         Cr2::read().as_u64(),
98         stack_frame
99     );
100 }
101
102 pub extern "x86-interrupt" fn x87_floating_point_handler(
103     stack_frame: InterruptStackFrame,
104 ) {
105     panic!("EXCEPTION: x87 FLOATING POINT\n{:#?}", stack_frame);
106 }
107
108 pub extern "x86-interrupt" fn alignment_check_handler(
109     stack_frame: InterruptStackFrame,
110     error_code: u64,
```

```

109 ) {
110     panic!(
111         "EXCEPTION: ALIGNMENT CHECK, ERROR_CODE: 0x{:016x}\n{:#?}",
112         error_code, stack_frame
113     );
114 }
115
116 pub extern "x86-interrupt" fn machine_check_handler(
117     stack_frame: InterruptStackFrame,
118 ) -> ! {
119     panic!("EXCEPTION: MACHINE CHECK\n{:#?}", stack_frame);
120 }
121
122 pub extern "x86-interrupt" fn simd_floating_point_handler(
123     stack_frame: InterruptStackFrame,
124 ) {
125     panic!("EXCEPTION: SIMD FLOATING POINT\n{:#?}", stack_frame);
126 }

```

7.2 q2

? Question


😬 你如何定义用于计数的 COUNTER，它能够做到线程安全吗？如果不能，如何修改？

定义的 COUNTER 是一个原子计数器，使用 AtomicUsize 类型来实现线程安全的计数。AtomicUsize 提供了原子操作的方法，如 `fetch_add` 和 `load`，可以在多线程环境中安全地进行加法和读取操作。代码如下：

```

1  static COUNTER: AtomicU64 = AtomicU64::new(0);
2
3  #[inline]
4  pub fn read_counter() -> u64 {
5      // FIXME: load counter value
6      COUNTER.load(Ordering::SeqCst)
7  }
8
9  #[inline]
10 pub fn inc_counter() -> u64 {
11     // FIXME: read counter value and increase it

```

 Rust

```
12     COUNTER.fetch_add(1, Ordering::SeqCst)
13 }
```

7.3 q3

? Question

😓 操作 APIC 时存在大量比特操作，尝试结合使用 `bitflags` 和 `bit_field` 来定义和操作这些寄存器的值，从而获得更好的可读性。

代码如下：

```
1  /// 用枚举表示 LAPIC 寄存器偏移 Rust
2  #[repr(u32)]
3  pub enum LpicRegister {
4      SVR      = 0xF0, // Spurious Interrupt Vector Register
5      TDCR     = 0x3E0, // Timer Divide Configuration Register
6      TICR     = 0x380, // Timer Initial Count Register
7      LvtTimer = 0x320, // LVT Timer Register
8      LvtLInt0 = 0x350, // LVT LINT0 Register
9      LvtLInt1 = 0x360, // LVT LINT1 Register
10     LvtPCINT = 0x340, // LVT Performance Counter Interrupt Register
11     LvtError = 0x370, // LVT Error Register
12     ESR      = 0x280, // Error Status Register
13     ICRLow   = 0x300, // Interrupt Command Register Low
14     ICRHigh  = 0x310, // Interrupt Command Register High
15     EOI      = 0x00B0, // End-of-Interrupt Register
16     TPR      = 0x80,   // Task Priority Register
17 }
18 // 使用 bitflags 定义 SVR 寄存器的标志位
19 bitflags! {
20     pub struct SvrFlags: u32 {
21         const ENABLE      = 1 << 8;
22         const VECTOR_MASK = 0xFF;
23     }
24 }
25 // 使用 bitflags 定义 LVT Timer 寄存器的标志位
26 bitflags! {
27     pub struct LvtTimerFlags: u32 {
28         const MASKED      = 1 << 16;
29         const PERIODIC     = 1 << 17;
```

```

30     const VECTOR_MASK = 0xFF;
31 }
32 }
33 bitflags! {
34     pub struct LvtErrorFlags: u32 {
35         const MASKED      = 1 << 16;
36         const VECTOR_MASK = 0xFF;
37     }
38 }

```

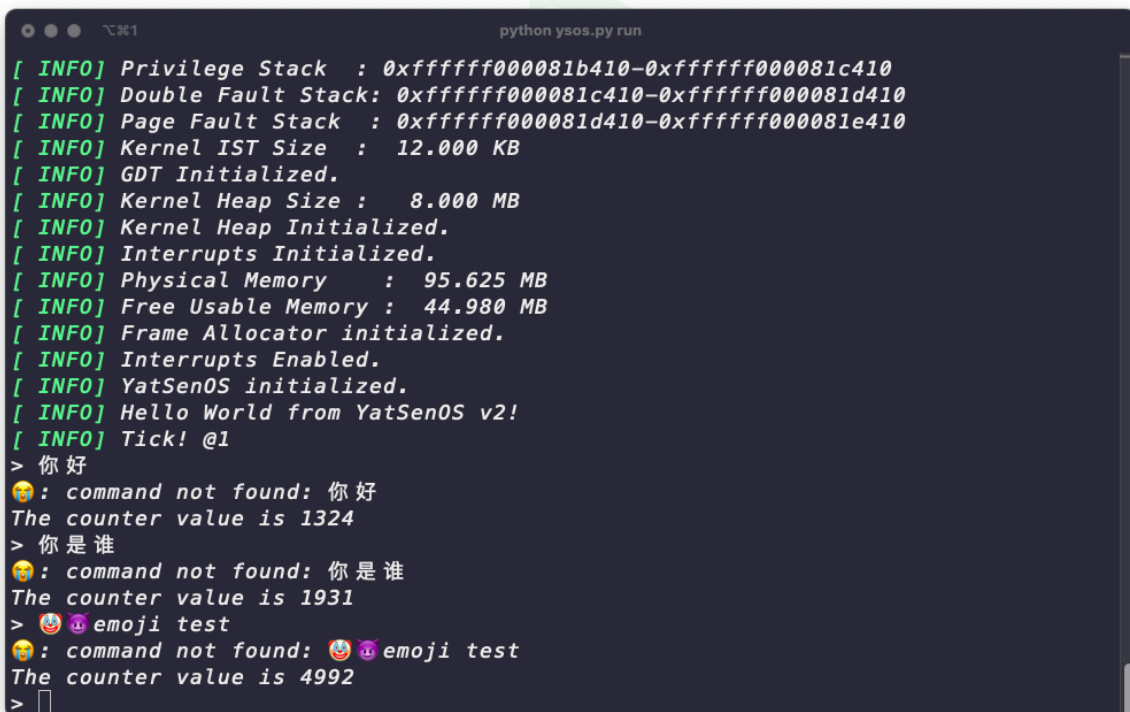
7.4 q4

? Question

🤔 你的串口输入驱动是否能正确的处理中文甚至 emoji 输入？如何能够正确处理？

可以采用 UTF-8 编码来处理中文和 emoji 输入。通过将输入的字节流解析为 UTF-8 字符串，可以正确处理这些字符。

效果如下：



```

python ysos.py run
[ INFO] Privilege Stack : 0xffffffff000081b410-0xffffffff000081c410
[ INFO] Double Fault Stack: 0xffffffff000081c410-0xffffffff000081d410
[ INFO] Page Fault Stack : 0xffffffff000081d410-0xffffffff000081e410
[ INFO] Kernel IST Size : 12.000 KB
[ INFO] GDT Initialized.
[ INFO] Kernel Heap Size : 8.000 MB
[ INFO] Kernel Heap Initialized.
[ INFO] Interrupts Initialized.
[ INFO] Physical Memory : 95.625 MB
[ INFO] Free Usable Memory : 44.980 MB
[ INFO] Frame Allocator initialized.
[ INFO] Interrupts Enabled.
[ INFO] YatSenOS initialized.
[ INFO] Hello World from YatSenOS v2!
[ INFO] Tick! @1
> 你好
🤔: command not found: 你好
The counter value is 1324
> 你是谁
🤔: command not found: 你是谁
The counter value is 1931
> 🤔🤔emoji test
🤔: command not found: 🤔🤔emoji test
The counter value is 4992
> 

```

图 7.4.2: 串口输入驱动处理中文和 emoji