

## 实验报告 2

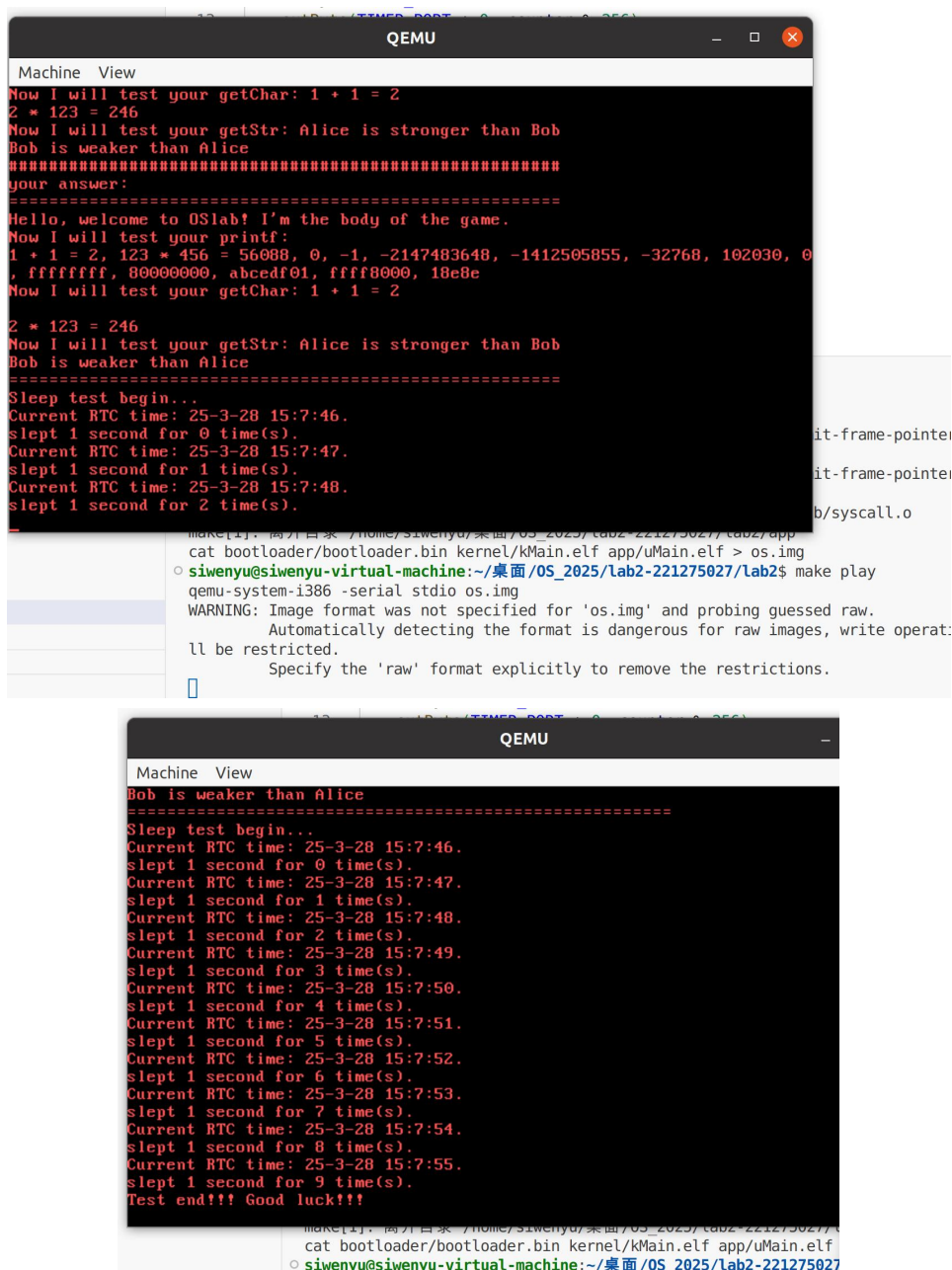
221275207 喻思文

[221275027@smail.nju.edu.cn](mailto:221275027@smail.nju.edu.cn)

### 一、实验进度

中断机制、键盘按键回显、printf()、getChar()、getStr()、sleep()、now()函数均完成。

### 二、运行截图



```
QEMU
Machine View
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2

2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
=====
Sleep test begin...
Current RTC time: 25-3-28 15:7:46.
slept 1 second for 0 time(s).
Current RTC time: 25-3-28 15:7:47.
slept 1 second for 1 time(s).
Current RTC time: 25-3-28 15:7:48.
slept 1 second for 2 time(s).
-
make[1]: 离开目录 /home/Siwenyu/桌面/OS_2025/lab2-221275027/lab2/app
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
Siwenyu@Siwenyu-virtual-machine:~/桌面/OS_2025/Lab2-221275027/Lab2$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operat:
ll be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

Bob is weaker than Alice
=====
Sleep test begin...
Current RTC time: 25-3-28 15:7:46.
slept 1 second for 0 time(s).
Current RTC time: 25-3-28 15:7:47.
slept 1 second for 1 time(s).
Current RTC time: 25-3-28 15:7:48.
slept 1 second for 2 time(s).
Current RTC time: 25-3-28 15:7:49.
slept 1 second for 3 time(s).
Current RTC time: 25-3-28 15:7:50.
slept 1 second for 4 time(s).
Current RTC time: 25-3-28 15:7:51.
slept 1 second for 5 time(s).
Current RTC time: 25-3-28 15:7:52.
slept 1 second for 6 time(s).
Current RTC time: 25-3-28 15:7:53.
slept 1 second for 7 time(s).
Current RTC time: 25-3-28 15:7:54.
slept 1 second for 8 time(s).
Current RTC time: 25-3-28 15:7:55.
slept 1 second for 9 time(s).
Test end!!! Good luck!!!

make[1]: 离开目录 /home/Siwenyu/桌面/OS_2025/lab2-221275027/lab2/app
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf
Siwenyu@Siwenyu-virtual-machine:~/桌面/OS_2025/Lab2-221275027
```

实际测试中，getChar()和 getStr()使用别的字符依然运行通过。backspace 键有效。

### 三、实验思路

#### (一) 中断机制

在 `kernal/kernal/doIrq.S` 中加入将 `irqKeyboard` 的中断向量号 `0x21` 压入栈。在 `kernal/kernal/idt.c` 进行一系列初始化。初始化 `interrupt gate`、`trap gate`、`idt` 表。两个门的字段如下：

offset_15_0	中断处理函数地址的低 16 位
offset_31_16	中断处理函数地址的高 16 位
segment	代码段选择子（需左移 3 位，因低 3 位是特权级和 TI 标志）
pad0	保留字段，设为 0
system	设为 FALSE 表示这是一个中断/陷阱门（非系统门）
type	设为 <code>INTERRUPT_GATE_32</code> 表示 32 位中断门
privilege_level	描述符特权级（DPL），控制哪些特权级代码可触发此中断
present	设为 TRUE 表示该描述符有效

`idt` 表设置如下，其中定时器和系统调用设置为中断门，其余设置为陷阱门，中断号已给出：

```
--
67      // TODO: 参考上面第48行(setTrap)代码填好剩下的表项
68      setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
69      setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, DPL_KERN);
70      setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent, DPL_KERN);
71      setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault, DPL_KERN);
72      setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
73      setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
74      setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck, DPL_KERN);
75      setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException, DPL_KERN);
76      setTrap(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
77      setIntr(idt + 0x20, SEG_KCODE, (uint32_t)irqTimer, DPL_KERN); // ?
78      setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
79
```

在 `kernal/kernal/irqHandle.c` 中 `irqHandle` 函数中进行中断程序处理：首先将数据段寄存器 `DS` 设置为内核数据段选择子（`SEG_KDATA`）。然后不同中断类型处理：

异常（如 `0xd`）：需处理错误状态（如修复页错误）。

硬件中断（如 `0x20-0x21`）：快速响应设备请求。

软中断（如 `0x80`）：实现系统调用接口。

## （二）键盘回显

在 `kernal/kernal/irqHandle.c` 的 `KeyboardHandle` 函数中，处理逻辑如下：

1. 键盘输入获取：调用 `getKeyCode()` 获取当前按键的扫描码。
2. 退格键处理：当检测到退格键(扫描码 `0xe`)时，检查当前光标不在行首 (`displayCol > 0`)，检查缓冲区指针是否有效(`bufferTail > tail`)，然后将光标位置回退一列(`displayCol--`)，并且回退缓冲区指针(`bufferTail--`)，最后在显存对应位置写入空格字符，实现字符擦除效果
3. 回车键处理：当检测到回车键(扫描码 `0x1c`)时，在缓冲区存入换行符，光标移动到下一行行首，重置行首标记，如果光标到达屏幕底部(第 25 行)，则执行屏幕滚动操作。
4. 普通字符处理：对于可打印字符(扫描码小于 `0x81`)，将扫描码转换为 ASCII 字符，并在缓冲区存储该字符，在屏幕当前光标位置显示字符。然后处理光标移动和自动换行，同样地，当到达屏幕底部时执行滚动操作

## （三）printf()

核心实现在 `lib/syscall.c` 中。主要实现如下：

1. 遍历格式字符串：循环遍历 `format` 字符串，直到遇到字符串结束符 `\0`。
2. 处理普通字符：将当前字符 `format[i]` 存入 `buffer`，并递增 `count`。

如果 `count` 达到 `MAX_BUFFER_SIZE`，调用 `syscall` 将缓冲区内容写入标准输出 (`STD_OUT`)，然后重置 `count` 为 0。

3. 处理格式说明符 (`'%'`)：

如果当前字符是 `%`，则进入格式说明符处理逻辑，回退 `count`（因为 `%` 不需要输出），移动到下一个字符，移动 `paraList` 指针以指向下一个参数 (`paraList+=sizeof(format)`)，根据 `%` 后的字符（如 `d`、`x`、`s`、`c`）调用相应的转换函数。

`%d`：调用 `dec2Str` 将整数转换为字符串。

`%x`：调用 `hex2Str` 将十六进制数转换为字符串。

`%s`：调用 `str2Str` 直接复制字符串。

`%c`：直接将字符存入缓冲区。

`%%`：输出 `%` 本身。

4. 刷新剩余缓冲区：如果循环结束后 `count` 不为 0，说明缓冲区还有未输出的内容，调用 `syscall` 将其写入标准输出。

`printf` 过程依赖于 `kernal/kernal/irqHandle.c` 中的 `sysPrint` 函数，主要逻辑如下：

将用户数据段的段选择子保存到 `sel` 中，用于后续访问用户空间的数据。从陷阱帧的 `edx` 寄存器中获取用户空间传递的字符串地址，并转换为 `char*` 类型。从陷阱帧的 `ebx` 寄存器中获取字符串的长度。通过内联汇编将 `sel`（用户数据段选择子）加载到 `es` 段寄存器中，以便后续访问用户空间的数据。

下面处理字符，通过 `for` 循环逐个处理字符串中的字符：从用户空间的字符串地址读取一个字节（字符）到 `character` 中。使用 `es` 段寄存器确保正确访问用户空间数据。

如果字符是换行符 `\n`：光标移动到下一行，回到行首。如果光标超出屏幕底部，滚屏，并将光标移动到最后一行的行首。

如果是普通字符：将字符与属性（`0x0c`，黑底红字）组合成显存数据（低字节是字符，高字节是属性），计算字符在显存中的位置（显存是线性地址，每行 80 字符，每个字符占 2 字节），将字符数据写入显存（`0xb8000` 是文本模式显存的起始地址），光标右移。如果光标超出行尾，换行并滚屏。

#### （四）`getChar()`、`getStr()`

实现过程在 `kernal/kernal/irqHandle.c` 中的 `sysGetChar()`、`sysGetStr()` 中，在 `lib/syscall.c` 中通过 `syscall` 调用 `sysRead` 从而使用他们。

- `sysGetChar()` 逻辑：首先获取键盘输入；然后将字符显示到屏幕，这里使用内联汇编将字符数据写入显存，然后更新光标位置，与前面思路基本一致。这里要处理回车键（换行），如果检测到回车键，换行，并检查是否需要滚动屏幕，更新光标位置并退出循环。将获取的字符存入中断帧的 `eax` 寄存器。

- `sysGetStr()` 逻辑：读取字符直到遇到换行符 `\n` 或达到最大长度 `size`。

如果当前键盘缓冲区位置 `keyBuffer[i]` 为空，则使用 `enableInterrupt()` 允许中断，让其他进程运行或等待键盘输入。一旦 `keyBuffer[i]` 不为空（有键盘输入），将 `keyBuffer[i]` 的值赋给 `tpc`。调用 `disableInterrupt()` 禁用中断，防止在复制字符时被其他中断干扰。

接着将字符写入用户空间。通过汇编指令设置 `es` 段寄存器为用户数据段，以便后续写入用户空间。将键盘缓冲区 `keyBuffer` 中的字符逐个写入用户空间指针 `str` 指向的位置。将 `keyBuffer[p]` 的值写入 `str + p - bufferHead` 的地址，并在字符串末尾写入终止符 `\0 (0x00)`，确保字符串以 `null` 结尾。

### （五）`sleep()`

计时器已在 `kernal/kernal/timer.c` 中实现，`100Hz` 频率（`10ms` 周期），因此将传入的毫秒数乘 `100`。通过每次时间单位到达后，循环递减 `time`，直到 `time` 变为负数，从而实现延时。调用系统调用 `SYS_TIME`，并传入 `SET_TIME_FLAG` 操作，可能是向内核请求设置一个时间标志（用于开始计时）。通过 `GET_TIME_FLAG` 操作轮询检查时间标志是否被清除（即是否到达一个时间单位）。

### （六）`now()`

具体实现在 `kernal/kernal/irqHandle.c` 中自己构建了 `sysNowTime` 函数，`lib/syscall.c` 中通过 `syscall` 调用。

`sysNowTime` 函数实现如下：将用户数据段的加载扇区加载到 `es` 寄存器中，以便后续通过 `es` 访问用户空间的内存。然后读取时间信息，调用 `outByte` 函数，向 `I/O` 端口 `0x70`（存储时间信息的特定寄存器）写入要读取的地址（`0x00` 是秒，`0x02` 是分钟，`0x04` 是小时，`0x07` 是天，`0x08` 是月，`0x09` 是年）。

调用 `inByte` 函数，从 `I/O` 端口 `0x71` 读取一个字节的的数据，用于读取或写入之前通过 `0x70` 端口选定的寄存器的值。

接着将读入的 `BCD` 格式的时间转换为十进制。再对已读入的各个数据进行时区转换（`RTC` 给的是 `UTC` 时间，北京时间是 `UTC+8`）通过 `es` 段寄存器将时间信息写入用户空间指针 `time_p` 指向的 `TimeInfo` 结构体（这里重新按照 `lib/lib.h` 构建的）。

## 四、思考题

### （一）什么是 IDT，为什么需要 IDT？IDT 相比 IVT 有什么优势

#### • IDT 的定义

在计算机体系结构中，IDT 是中断描述符表（Interrupt Descriptor Table）的缩写。它是一个数据结构，用于在 x86 架构的保护模式下处理中断和异常。IDT 由多个 8 字节的描述符（称为门）组成，每个门指向一个中断服务例程（ISR）的地址。中断描述符表寄存器（IDTR）保存了 IDT 的基地址和限制。

#### • 需要 IDT 的原因

IDT 在保护模式下是必要的，因为它提供了中断和异常处理的机制。现代操作系统几乎完全在保护模式下运行，因此需要使用 IDT 来处理硬件中断、软件异常以及系统调用等。此外，IDT 还提供了对中断处理的保护机制，例如通过中断门和陷阱门来控制对中断服务例程的访问，从而增强了系统的安全性和稳定性。

#### • IDT 相比 IVT 的优势

1. 支持更多中断类型：IVT 在实模式下通常固定为 256 个中断向量，而 IDT 在保护模式下可以支持更多的中断类型，具体数量由 IDT 的大小决定。

2. 增强的访问控制：IDT 中的门描述符（如中断门和陷阱门）可以指定访问权限，从而增强了系统的安全性和稳定性。

3. 支持保护模式：IDT 是为保护模式设计的，能够更好地支持现代操作系统的内存保护和多任务处理功能。

### （二）接下来的问题是，哪些内容表征了任务 A 的状态？CPU 又应该将它们保存到哪里去？

#### • 表征任务 A 状态的内容主要包括以下寄存器中的信息：

1. 程序计数器（PC）：保存下一条将要执行的指令的内存地址。

2. 指令寄存器（IR）：保存当前正在执行的指令。

3. 程序状态字寄存器（PSW）：包含反映当前运算状态及程序工作方式的各种标志位，如进位标志位（CF）、结果为零标志位（ZF）、符号标志位（SF）、

溢出标志位（OF）等。

4. 通用寄存器：用于保存操作数、地址和中间结果等。

5. 栈指针寄存器：指向当前任务的栈顶位置，用于保存函数调用和中断处理时的现场信息。

- 保存位置：

在正常执行过程中，CPU 会将这些状态信息保存在自身的寄存器中，以便快速访问和操作。然而，在发生中断或进行任务切换时，为了保证任务能够正确地恢复执行，CPU 需要将当前任务的状态信息保存到内存中。通常，这些信息会被保存到任务的进程控制块中，或者在中断处理时保存到中断栈中。这样，在中断返回或切换回任务时，CPU 可以从内存中恢复这些状态信息，继续执行任务。

### （三）什么是 TSS，为什么需要 TSS？TSS 的结构是如何？

- TSS 的定义

在计算机体系结构中，TSS 是任务状态段（Task State Segment）的缩写。它是一个特殊的数据结构，用于在 x86 架构的保护模式下保存任务（进程或线程）运行时的状态信息。每个任务都有自己的 TSS，它包含了任务切换时需要保存和恢复的上下文信息。

- 为什么需要 TSS

在多任务操作系统中，当发生任务切换时，需要保存当前任务的状态，以便稍后能够正确地恢复执行。TSS 提供了硬件级别的任务上下文切换能力，能够快速保存和恢复任务的状态，从而提高系统的效率和稳定性。此外，TSS 还支持特权级切换时的安全性，确保在从低特权级切换到高特权级时能够正确地访问内核栈等关键资源。

- TSS 的结构

TSS 的结构通常包括以下字段：



1. 旧任务链接：保存前一个任务的 TSS 选择子，用于任务嵌套。
2. ESP0、SS0：用于特权级切换时的栈指针和栈段选择子（从低特权级切换到高特权级）。
3. 程序计数器（EIP）：保存下一条将要执行的指令的内存地址。
4. 段寄存器（CS、DS、ES、FS、GS、SS）：保存各段寄存器的值。
5. 通用寄存器（EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI）：保存通用寄存器的值。
6. 指令指针（EIP）：保存下一条指令的地址。
7. 程序状态字（EFLAGS）：保存程序状态字寄存器的值。
8. I/O 映射基地址：用于决定当前任务是否可以访问特定硬件端口。
9. LDT 段选择子：指向当前任务的局部描述符表（LDT）。
10. 其他控制信息：如调试标志等。

（四）IA-32 提供了 4 个特权级，但 TSS 中只有 3 个堆栈位置信息，分别用于 ring0, ring1, ring2 的堆栈切换。为什么 TSS 中没有 ring3 的堆栈信息？

1. 特权级切换的方向性：在 x86 架构中，TSS 主要用于在特权级切换时保存和恢复任务的状态。特权级切换通常是从低特权级（如 ring3）到高特权级（如 ring0）进行的，例如在系统调用或中断处理时。而从高特权级返回低特权级时，通常不需要切换堆栈，因为此时的堆栈信息已经被保存在高特权级的堆栈中。因此，TSS 中只需要为高特权级（ring0、ring1、ring2）的堆栈提供位置信息。

2. 硬件堆栈切换机制：当从低特权级切换到高特权级时，CPU 会自动从 TSS 中读取相应特权级的堆栈信息（如 SS0 和 ESP0），并切换到新的堆栈。这种硬件机制确保了在进入高特权级时，能够安全地保存低特权级的上下文信息。而当从高特权级返回低特权级时，CPU 会从高特权级的堆栈中恢复低特权级的上下文信息，并不需要再次访问 TSS。

3. 软件实现的灵活性：现代操作系统通常采用软件方式来管理任务切换和堆栈信息，而不是完全依赖硬件提供的 TSS 机制。在这些系统中，ring3 的堆栈信息可以通过其他方式（如内核栈）进行管理和保存，因此不需要在 TSS 中为 ring3 单独预留堆栈位置。



（五）我们在使用 `eax, ebx, ecx, edx, esi, edi` 前将寄存器的值保存到了栈中，如果去掉保存和恢复的步骤，从内核返回之后会不会产生不可恢复的错误？

可能会，理由如下：

可能产生如下问题

1. 用户态程序的寄存器值被破坏：如果内核代码修改了用户态程序正在使用的寄存器（如 `ebx`、`esi`、`edi`）值，而没有恢复，用户态程序可能会因为寄存器值的不一致而导致错误，甚至崩溃。

2. 系统稳定性问题：如果内核代码修改了寄存器（如 `eax`、`ecx`、`edx`）的值，从内核返回用户态后，用户态程序可能会因为这些寄存器值的变化而出现问题。寄存器值的不正确可能导致系统调用返回时的参数错误，进而引发系统不稳定或其他不可预测的行为。

（六）尽管时钟中断程序的执行时间相较于 10 毫秒来说非常短暂，是否考虑过更高精度的 `sleep` 实现方式？

将 `kernal/kernal/timer.c` 中的 `HZ` 值（代表时钟频率）改为 100 以上（如 1000，执行时间 1ms），就是更高的频率，`sleep` 精度更高。

（七）该如何描述系统中断的处理过程？为什么要有系统中断的机制？

• 系统中断的处理过程

1. 中断请求产生：当外部设备（如键盘、鼠标、磁盘等）或内部事件（如时钟中断、异常等）发生时，会产生一个中断信号，通知 CPU 有紧急任务需要处理。

2. 中断识别：CPU 在接收到中断信号后，会暂停当前正在执行的任务，保存当前任务的状态（如程序计数器、寄存器等），然后通过中断控制器识别中断的类型号，根据中断类型号找到对应的中断服务程序（ISR）的入口地址。

3. 中断处理：CPU 跳转到中断服务程序的入口地址，开始执行中断处理程序。中断服务程序通常会执行以下操作：

- 1) 保存被中断任务的现场信息（如寄存器值等）到堆栈或其他安全位置。
- 2) 对中断事件进行具体处理，如读取键盘输入、处理磁盘数据等。

3) 更新相关硬件状态或软件标志,以通知操作系统或其他部分中断已处理。

4) 检查是否有更高优先级的中断需要处理,如果有,则转去处理更高优先级的中断。

4. 中断返回:当中断服务程序执行完毕后,CPU 需要恢复被中断任务的现场信息(从堆栈中恢复寄存器值等),然后返回到被中断任务的断点处,继续执行被中断的任务。

#### • 为什么要有系统中断机制

1. 提高系统效率:系统中断机制允许 CPU 在执行当前任务时,能够及时响应外部设备或内部事件的请求,而不需要 CPU 不断地轮询设备状态,从而提高了系统的整体效率和资源利用率。

2. 实现异步操作:中断机制使得设备和 CPU 可以并行工作,设备可以在准备好数据后再通过中断通知 CPU 进行处理,实现了异步操作,提高了系统的并发处理能力。

3. 增强系统响应能力:通过中断机制,系统能够快速响应各种紧急事件和异常情况,如硬件故障、用户输入等,及时进行处理,提高了系统的实时性和可靠性。

4. 简化程序设计:中断机制为程序设计提供了便利,使得程序可以更加模块化和灵活。程序可以通过设置中断服务程序来处理特定的事件,而不需要在主程序中不断地检查和处理各种情况。