

1

因为任务切换通常是通过中断处理程序来实现的，而不是通过函数调用来实现的。在中断处理程序中，处理器会自动将当前指令的地址压入内核栈中，然后跳转到中断处理程序的入口点。因此，在任务切换过程中，不需要显式地使用 `call` 指令来调用中断处理程序。

在任务切换的最后，使用 `ret` 指令返回到下一个任务的代码执行点。这里的 `ret` 指令会从内核栈中弹出返回地址，并将其存入处理器的 `EIP` 寄存器中，从而实现跳转到下一个任务的代码。

2

该函数的作用是将任务的上下文（包括通用寄存器、标志寄存器和栈指针等）初始化到任务内核栈中。

1. 首先，使用指针后缀运算符 (`--`) 将 `stk` 指向的内核栈指针地址向下移动一个单元，然后将 `0x08`（即内核代码段选择子）压入内核栈中，作为任务切换返回后的代码段选择子。
2. 接下来，再次使用指针后缀运算符将 `stk` 指向的内核栈指针地址向下移动一个单元，然后将任务函数的地址压入内核栈中，作为任务切换返回后的代码执行点。
3. 继续使用指针后缀运算符将 `stk` 指向的内核栈指针地址向下移动一个单元，然后将 `0x0202`（即 `EFLAGS` 寄存器的值）压入内核栈中。
4. 接下来，依次将 `EAX`、`ECX`、`EDX`、`EBX`、`ESP`、`EBP`、`ESI` 和 `EDI` 等通用寄存器的初始值压入内核栈中。
5. 最后，将 `stk` 指向的内核栈指针地址所指向的内存单元的值设置为 `0x77777777`，这是任务切换返回后处理器的 `EDI` 寄存器的初始值

3

`stack[STACK_SIZE]` 是分配给当前任务的栈

`BspContextBase[STACK_SIZE]` 作为所有任务启动前的任务栈。当调用 `myTask0` 时，此时 `prev_task` 为空值，需要从 `BspContextBase` 开始中断程序，跳转执行 `myTask0`

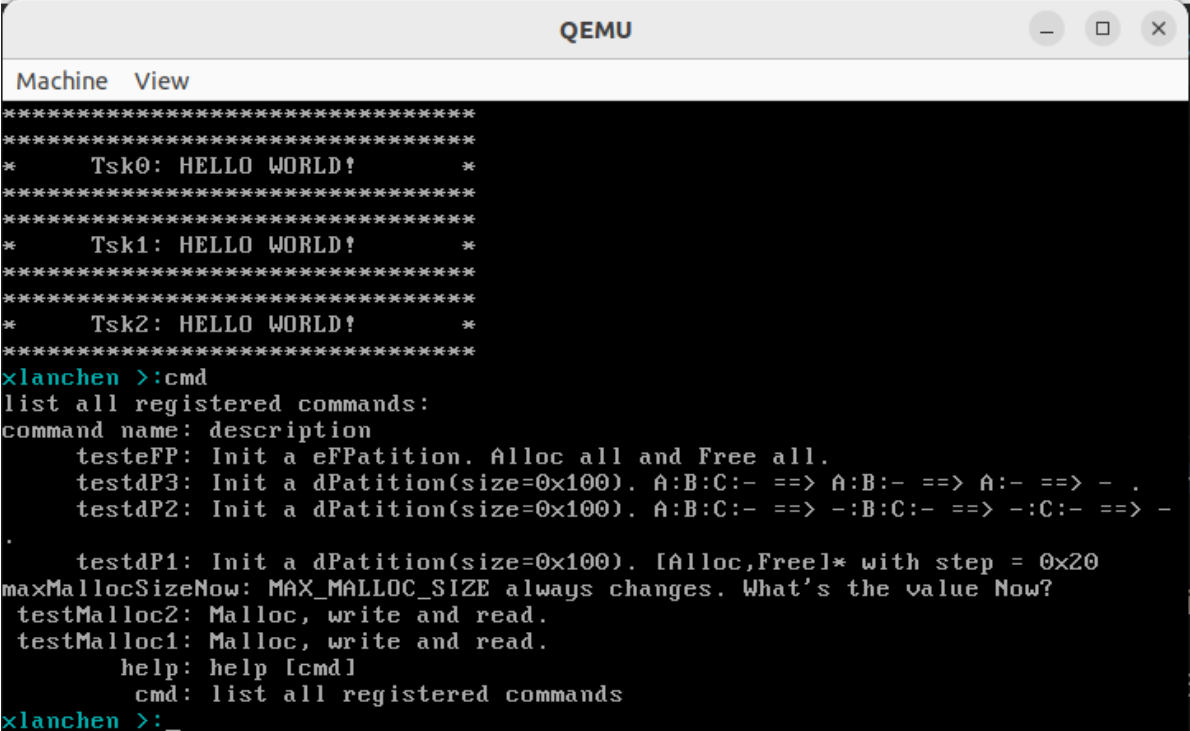
4

二级指针

`prevTSK StackPtrAddr`是指针的指针，被存入 `eax` 寄存器

指令 `movl %esp, (%eax)` 将当前任务的栈指针 `%esp` 保存到 `prevTSK_StackPtr` 指向的内存地址 `(%eax)` 中

实验结果·



```
Machine View
*****
*****
*      Tsk0: HELLO WORLD!      *
*****
*****
*      Tsk1: HELLO WORLD!      *
*****
*****
*      Tsk2: HELLO WORLD!      *
*****
xlanchen >:cmd
list all registered commands:
command name: description
    testeFP: Init a eFPatition. Alloc all and Free all.
    testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
    testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
.
    testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
    testMalloc2: Malloc, write and read.
    testMalloc1: Malloc, write and read.
        help: help [cmd]
        cmd: list all registered commands
xlanchen >:

```