

操作系统实验作业——lab1

操作系统实验作业——lab1

练习1：理解内核启动中的程序入口操作

请描述操作系统的内核启动流程：

说明指令 `la sp, bootstacktop` 完成了什么操作，目的是什么：

`tail kern_init` 完成了什么操作，目的是什么？

练习2：完善中断处理（需要编程）

实现代码

输出结果

实现过程和定时器中断中断处理的流程：

扩展练习 Challenge1：描述与理解中断流程

1. 在 uCore 中处理中断和异常的流程如下：

2. `mov a0, sp` 的目的

3. `SAVE_ALL` 中寄存器保存在栈中的位置是什么确定的

4. 对于任何中断，`__alltraps` 中都需要保存所有寄存器吗

扩展练习 Challenge2：理解上下文切换机制

在 `trapentry.S` 中汇编代码 `csw sscratch, sp; csrrw s0, sscratch, x0` 实现了什么操作，目的是什么？

`save all` 里面保存了 `stval` `scause` 这些 `csr`，而在 `restore all` 里面却不还原它们？那这样 `store` 的意义何在呢？

扩展练习 Challenge3：完善异常中断

代码如下

输出结果

练习1：理解内核启动中的程序入口操作

阅读 `kern/init/entry.S` 内容代码，结合操作系统内核启动流程，说明指令 `la sp, bootstacktop` 完成了什么操作，目的是什么？`tail kern_init` 完成了什么操作，目的是什么？

请描述操作系统的内核启动流程：

（1）硬件初始化完成以后，计算机会加载并启动内核。首先需要从磁盘中读取内核映像，内核映像包含了操作系统的所有代码和数据，它主要通过磁盘上的文件系统来提供；

（2）将内核映像进行解压：内核映像通常以压缩的形式提供，需要对其进行解压以后才可以被内核读取；

（3）跳转到内核的入口点：内核的入口点是一个特殊的函数，它是内核运行的起始点。当内核启动时，它会跳转到这个入口点开始执行。

说明指令la sp,bootstacktop完成了什么操作，目的是什么：

整个而言，qemu的模拟器其实是提供了一个RISC-V的cpu，物理内存以及总线功能。完成这项工作的是bootloader，主要负责开机以及将操作系统加载到内核。其中在qemu中提供了固件openSBI来完成这项工作，而kern/init/entry.S就是内核的入口点。在riscv上电时，会进行CPU自检，然后跳转到bootloader处执行。bootloader设置好kernel的运行环境后，从硬盘加载kernel到内存，最后再跳转到kernel入口地址。我们采用的bootloader为OpenSBI，被加载到0x80000000地址，OpenSBI探测好外设并初始化内核的环境变量后，加载内核到0x80200000地址，最后再跳转到0x80200000地址。这行代码，是把bootstacktop(也就是内核的栈顶部)的地址加载到sp（堆栈指针）寄存器中，主要目的是为后续的内核初始化准备好堆栈环境，在刚开始上电的时候，系统还没有高级程序语言的开发环境，所以需要依赖汇编语言，而堆栈环境的设置就可以保证函数调用，参数传递等的顺利进行。

tail kern_init完成了什么操作，目的是什么？

指令tail kern_init是一个尾调用的指令，它会直接跳转到汇编代码中标签kern_init所在的位置进行执行，在整个过程中不会保存返回地址，后续无需返回。主要作用是进行内核的初始化操作，并通过尾调用的方式抛弃当前的堆栈帧，充分利用栈空间资源。该函数时内核的主函数，用于完成内核的初始化的操作。

练习2：完善中断处理（需要编程）

请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字“100 ticks”，在打印完10行后调用sbi.h中的shut_down()函数关机。

要求完成问题1提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程和定时器中断中断处理的流程。实现要求的部分代码后，运行整个系统，大约每1秒会输出一行“100 ticks”，输出10行。

实现代码

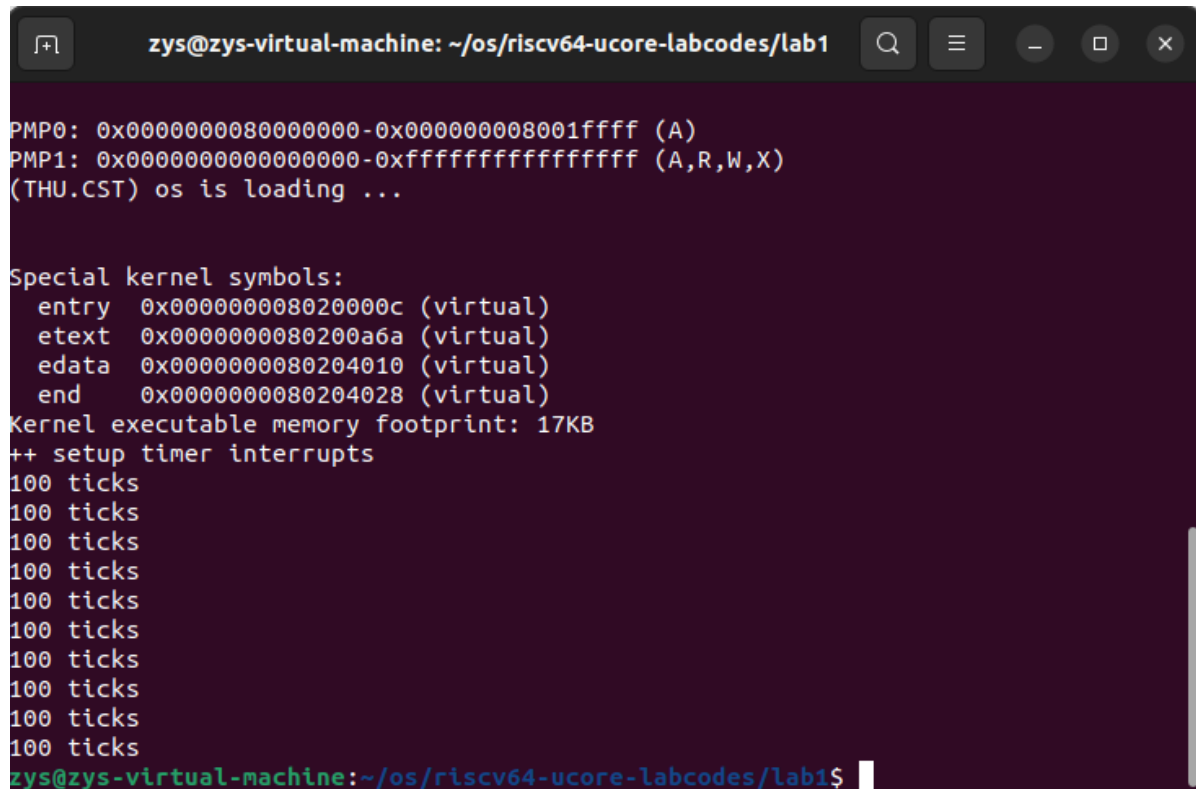
```
1      case IRQ_S_TIMER:
2          // "All bits besides SSIP and USIP in the sip register are
3          // read-only." -- privileged spec1.9.1, 4.1.4, p59
4          // In fact, call sbi_set_timer will clear STIP, or you can
clear it
5          // directly.
6          // cprintf("Supervisor timer interrupt\n");
7          /* LAB1 EXERCISE2   YOUR CODE :  */
8          /*(1)设置下次时钟中断- clock_set_next_event()
9          *(2)计数器（ticks）加一
10         *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触
发了100次时钟中断，同时打印次数（num）加一
11         * (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关
机
12         */
13         clock_set_next_event();
14         ticks++;
```

```

15         if(ticks % TICK_NUM == 0){
16             print_ticks();//当计数器加到100的时候，我们会输出一个
           `100ticks`表示我们触发了100次时钟中断
17             num++; //打印次数+1
18             if(num==10){
19                 sbi_shutdown();
20             } //当打印次数为10时，调用<sbi.h>中的关机函数关机
21         }
22         break;

```

输出结果



```

zys@zys-virtual-machine: ~/os/riscv64-ucore-labcodes/lab1
PMP0: 0x0000000008000000-0x0000000008001fff (A)
PMP1: 0x0000000000000000-0xffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0x000000000802000c (virtual)
  etext 0x00000000080200a6a (virtual)
  edata 0x00000000080204010 (virtual)
  end   0x00000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
zys@zys-virtual-machine:~/os/riscv64-ucore-labcodes/lab1$

```

实现过程和定时器中断中断处理的流程：

一开始只设置一个时钟中断，之后每次发生时钟中断的时候，设置下一次的时钟中断。每次触发时钟中断的时候，我们会给一个计数器加一，并且设定好下一次时钟中断。当计数器加到100的时候，我们会输出一个 `100ticks` 表示我们触发了100次时钟中断。

产生一次时钟中断的执行流：`set_sbi_timer()`通过OpenSBI的时钟事件触发一个中断，跳转到 `kern/trap/trapentry.S` 的 `__alltraps` 标记 -> 保存当前执行流的上下文，并通过函数调用，切换为 `kern/trap/trap.c` 的中断处理函数 `trap()` 的上下文，进入 `trap()` 的执行流。切换前的上下文作为一个结构体，传递给 `trap()` 作为函数参数 -> `kern/trap/trap.c` 按照中断类型进行分发(`trap_dispatch()`, `interrupt_handler()`) -> 执行时钟中断对应的处理语句，累加计数器，设置下一次时钟中断 -> 完成处理，返回到 `kern/trap/trapentry.S` -> 恢复原先的上下文，中断处理结束。

扩展练习 Challenge1：描述与理解中断流程

回答：描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0, sp的目的是什么？SAVE_ALL中寄存器保存在栈中的位置是什么确定的？对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由。

1.在 uCore 中处理中断和异常的流程如下：

1. **异常的产生**：中断或异常是由硬件或软件引发的事件，可以是外部中断、系统调用、硬件故障等。这会导致处理器从当前执行的程序切换到相应的中断或异常处理程序。在本次实验中我们通过函数和内联汇编的形式出发中断和异常。

```
1  asm volatile("ebreak");//CAUSE_BREAKPOINT
2  asm volatile(".word 0x00000000");//CAUSE_ILLEGAL_INSTRUCTION
3
4  /* intr_enable - enable irq interrupt */
5  void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
```

1. **硬件响应**：当异常或中断发生时，处理器会自动执行特定的操作，以保护异常处理程序的执行。这通常包括将当前程序状态保存到寄存器中，以便稍后恢复执行。
2. **跳转到异常处理程序**：处理器会跳转到事先定义的异常处理程序的入口点。在 uCore 中，这是 __alltraps函数。idt_inti函数用于初始化中断描述符表（IDT），通过write_csr将sscratch寄存器设置为0，即告诉异常向量表，当前正在内核模式下执行异常处理，以确保在处理异常期间使用内核栈而不是用户栈。write_csr(stvec, &__alltraps)将stvec寄存器设置为__alltraps函数的地址，当异常或中断发生时，控制将跳转到该函数，从而开始执行异常处理程序。

```
1  void idt_init(void) {
2      extern void __alltraps(void);
3      /* Set sscratch register to 0, indicating to exception vector that
4         we are
5         * presently executing in the kernel */
6      write_csr(sscratch, 0);
7      /* Set the exception vector address */
8      write_csr(stvec, &__alltraps);
9  }
```

4. **保存寄存器状态**：在 __alltraps函数中，首先使用 SAVE_ALL 宏将寄存器的状态保存到内核栈中。这个操作的目的是保护当前进程的寄存器状态，以确保在中断处理完成后可以正确地恢复程序的执行。这个过程也称为上下文切换。

```
1  (gdb) x/10i $pc
2  => 0x80200526 <__alltraps+6>:  sd  zero,0(sp)
3      0x80200528 <__alltraps+8>:  sd  ra,8(sp)
4      0x8020052a <__alltraps+10>: sd  gp,24(sp)
5      0x8020052c <__alltraps+12>: sd  tp,32(sp)
6      0x8020052e <__alltraps+14>: sd  t0,40(sp)
7      0x80200530 <__alltraps+16>: sd  t1,48(sp)
8      0x80200532 <__alltraps+18>: sd  t2,56(sp)
```

```

9      0x80200534 <__alltraps+20>: sd s0,64(sp)
10     0x80200536 <__alltraps+22>: sd s1,72(sp)
11     0x80200538 <__alltraps+24>: sd a0,80(sp)
12 (gdb)
13     0x8020053a <__alltraps+26>: sd a1,88(sp)
14     0x8020053c <__alltraps+28>: sd a2,96(sp)
15     0x8020053e <__alltraps+30>: sd a3,104(sp)
16     0x80200540 <__alltraps+32>: sd a4,112(sp)
17     0x80200542 <__alltraps+34>: sd a5,120(sp)
18     0x80200544 <__alltraps+36>: sd a6,128(sp)
19     0x80200546 <__alltraps+38>: sd a7,136(sp)
20     0x80200548 <__alltraps+40>: sd s2,144(sp)
21     0x8020054a <__alltraps+42>: sd s3,152(sp)
22     0x8020054c <__alltraps+44>: sd s4,160(sp)
23 (gdb)
24     0x8020054e <__alltraps+46>: sd s5,168(sp)
25     0x80200550 <__alltraps+48>: sd s6,176(sp)
26     0x80200552 <__alltraps+50>: sd s7,184(sp)
27     0x80200554 <__alltraps+52>: sd s8,192(sp)
28     0x80200556 <__alltraps+54>: sd s9,200(sp)
29     0x80200558 <__alltraps+56>: sd s10,208(sp)
30     0x8020055a <__alltraps+58>: sd s11,216(sp)
31     0x8020055c <__alltraps+60>: sd t3,224(sp)
32     0x8020055e <__alltraps+62>: sd t4,232(sp)
33     0x80200560 <__alltraps+64>: sd t5,240(sp)
34 (gdb)
35     0x80200562 <__alltraps+66>: sd t6,248(sp)

```

4. **确定异常类型：**__alltraps函数会根据保存的异常号和异常参数确定引发异常的具体类型。

在本实验中通过interrupt_handler()和exception_handler()判断异常类型并调用相关处理函数。

```

1 void interrupt_handler(struct trapframe *tf) { //中断处理
2     intptr_t cause = (tf->cause << 1) >> 1;
3     switch (cause) {
4         .....
5     }
6 void exception_handler(struct trapframe *tf) { //异常处理
7     switch (tf->cause) {
8         .....
9     }

```

4. **执行异常处理程序：**根据异常类型，处理器会跳转到相应的异常处理程序。在这里，特定的异常处理逻辑将会执行，可能包括错误处理、中断处理、系统调用处理等。

```

1 Exception type: Illegal instruction
2 EPC: 0x80200048

```

4. **返回用户态**：一旦异常或中断处理完成，处理器会使用 `restore_all`宏从内核栈中还原之前保存的寄存器状态。这个操作将恢复之前中断或异常发生时的程序状态，以便程序可以继续执行。

```
1  __alltraps:
2      SAVE_ALL
3
4      move  a0, sp
5      jal  trap
6      # sp should be the same as before "jal trap"
7
8      .globl __trapret
9  __trapret:
10     RESTORE_ALL
11     # return from supervisor call
12     sret
```

```
1  (gdb) x/10i $pc
2  => 0x80200588 <__trapret>:  ld  s1,256(sp)
3      0x8020058a <__trapret+2>:  ld  s2,264(sp)
4      0x8020058c <__trapret+4>:  csrw  sstatus,s1//恢复中断/异常处理前
    的处理器状态。
5      0x80200590 <__trapret+8>:  csrw  sepc,s2//恢复中断/异常处理前的异
    常程序计数器值。
6      0x80200594 <__trapret+12>: ld  ra,8(sp)
7      0x80200596 <__trapret+14>: ld  gp,24(sp)
8      0x80200598 <__trapret+16>: ld  tp,32(sp)
9      0x8020059a <__trapret+18>: ld  t0,40(sp)
10     0x8020059c <__trapret+20>: ld  t1,48(sp)
11     0x8020059e <__trapret+22>: ld  t2,56(sp)
12  (gdb)
13     0x802005a0 <__trapret+24>: ld  s0,64(sp)
14     0x802005a2 <__trapret+26>: ld  s1,72(sp)
15     0x802005a4 <__trapret+28>: ld  a0,80(sp)
16     0x802005a6 <__trapret+30>: ld  a1,88(sp)
17     0x802005a8 <__trapret+32>: ld  a2,96(sp)
18     0x802005aa <__trapret+34>: ld  a3,104(sp)
19     0x802005ac <__trapret+36>: ld  a4,112(sp)
20     0x802005ae <__trapret+38>: ld  a5,120(sp)
21     0x802005b0 <__trapret+40>: ld  a6,128(sp)
22     0x802005b2 <__trapret+42>: ld  a7,136(sp)
23  (gdb)
24     0x802005b4 <__trapret+44>: ld  s2,144(sp)
25     0x802005b6 <__trapret+46>: ld  s3,152(sp)
26     0x802005b8 <__trapret+48>: ld  s4,160(sp)
27     0x802005ba <__trapret+50>: ld  s5,168(sp)
28     0x802005bc <__trapret+52>: ld  s6,176(sp)
29     0x802005be <__trapret+54>: ld  s7,184(sp)
30     0x802005c0 <__trapret+56>: ld  s8,192(sp)
31     0x802005c2 <__trapret+58>: ld  s9,200(sp)
```

```

32      0x802005c4 <__trapret+60>:   1d   s10,208(sp)
33      0x802005c6 <__trapret+62>:   1d   s11,216(sp)
34  (gdb)
35      0x802005c8 <__trapret+64>:   1d   t3,224(sp)
36      0x802005ca <__trapret+66>:   1d   t4,232(sp)
37      0x802005cc <__trapret+68>:   1d   t5,240(sp)
38      0x802005ce <__trapret+70>:   1d   t6,248(sp)
39      0x802005d0 <__trapret+72>:   1d   sp,16(sp)
40      0x802005d2 <__trapret+74>:   sret//从栈中恢复寄存器的值并从异常处理程序
      返回到被中断的代码

```

2.mov a0, sp的目的

至于问题中提到的 `mov a0, sp` 的目的是为了将当前的栈指针 `sp` 的值保存到寄存器 `a0` 中，通常是为了将异常处理程序的参数传递给其他函数或系统调用。

```

1      .globl __alltraps
2      .align(2)
3  __alltraps:
4      SAVE_ALL
5
6      move  a0, sp
7      jal  trap
8      # sp should be the same as before "jal trap"
9
10     .globl __trapret
11  __trapret:
12     RESTORE_ALL
13     # return from supervisor call
14     sret

```

3.SAVE_ALL中寄存器保存在栈中的位置是什么确定的

在 `SAVE_ALL` 中，寄存器保存在内核栈中的位置是由宏定义的。通常情况下，内核栈位于内核地址空间的某个位置，它的大小足够大，可以容纳所有需要保存的寄存器值。保存的寄存器值是按照特定的顺序依次压入栈中，以确保它们在栈上的布局是固定的，这样在 `RESTORE_ALL` 时，只需要根据相同的规则便可以对栈中的数据依次读取，然后存在寄存器中。

4.对于任何中断，__alltraps 中都需要保存所有寄存器吗

对于任何中断，`__alltraps` 中并不一定需要保存所有寄存器。具体需要保存哪些寄存器取决于中断类型和异常处理的需求。一般来说，至少需要保存通用寄存器、程序计数器、栈指针等，以便能够正确地还原进程的上下文。其他寄存器如浮点寄存器或向量寄存器，根据需要进行保存和还原。保存寄存器的目的是为了确保异常处理程序可以安全地执行并且后续能够正确地返回到用户态或继续执行中断处理。

扩增练习 Challenge2：理解上下文切换机制

回答：在trapentry.S中汇编代码 `csrw sscratch, sp; csrrw s0, sscratch, x0`实现了什么操作，目的是什么？`save all`里面保存了`stval scause`这些csr，而在`restore all`里面却不还原它们？那这样store的意义何在呢？

在trapentry.S中汇编代码 `csrw sscratch, sp; csrrw s0, sscratch, x0`实现了什么操作，目的是什么？

1. `csrw sscratch, sp`：表示将栈指针(`sp`)的值写入 `sscratch` 寄存器，主要目的是暂存当前的栈指针(`sp`)，`sscratch`寄存器通常用于保存一个临时的、可用于异常处理的值。在这里，将`sp`写入`sscratch`的目的是保存当前的栈指针，以便在异常处理过程中恢复。
2. `csrrw s0, sscratch, x0`：表示读取 `sscratch` 寄存器的值（之前暂存的栈指针）到 `s0`，并将 `x0`（零寄存器，永远为0）的值写入 `sscratch`。也就是说，切换 `sscratch` 和 `s0` 的值。这样做一方面恢复了之前保存的栈状态，并且给了 `sscratch` 一个新的值（这里是0），防止旧的栈指针信息被错误地使用。通常，在异常处理开始时，会将`sscratch`寄存器设置为零，以指示异常处理程序正在内核态执行然后在处理完成后，可以恢复`sscratch`寄存器的旧值，以确保异常处理程序能够正常返回。

save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢？

"SAVE_ALL" 和 "RESTORE_ALL" 这两个宏主要用于保存和恢复上下文，以防中断、异常或系统调用时改变了程序的执行状态。通常来讲，不同的异常处理程序会在其处理过程中修改这些CSR(control and status registers)，而保存这些寄存器的主要目的就是为了让异常处理程序返回到刚被打断的原有状态。

将 `stval`、`scause` 这些CSR的值保存到栈里，在异常处理引发新的异常等情况下是十分必要的。保存这些值的意义在于，确保在当前异常处理结束后，如果函数栈被切换回低级别函数时，这些CSR的值还能恢复到原来的状态。

而对于为何在 "RESTORE_ALL" 里面不还原这些寄存器的值，主要原因是：在处理异常或系统调用的过程中，我们的目标是修复或响应异常，而不是继续产生异常。也就是说，异常处理程序在执行完之后，一般不需要再次引起同样的异常。因此，即使在处理完异常后，`scause` 或 `stval` 的值已经改变，也没必要把它们恢复到引发异常时的原始值。

扩展练习Challenge3：完善异常中断

编程完善在触发一条非法指令异常 `mret`和，在 `kern/trap/trap.c`的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即"illegal instruction caught at 0x(地址)"，`"ebreak caught at 0x (地址)"`与"Exception type:illegal instruction"，"Exception type: breakpoint"。（

在init.c程序中，在kern_init()函数中，进行内核初始化，再进行idt_init()即异常表加载后我们添加内置汇编程序

分别来触发异常和中断如下：

```
1  __asm__ volatile("ebreak");// breakpoint
2  __asm__ volatile(".word 0x00000000");// illegal instruction
```

两段的用处如下

```
1  __asm__ volatile("ebreak");// breakpoint
```

这是一个用于触发断点异常的指令。在RISC-V架构中，ebreak指令用于引发一个断点异常，通常用于调试和异

常处理。当处理器执行到这个指令时，它会跳转到相应的异常处理程序，以便进行调试或其他必要的操作。

```
1  __asm__ volatile(".word 0x00000000");// illegal instruction
```

这是一个用于生成特定的机器码指令的内联汇编语句。在这种情况下，它生成的是一个机器码值为0的指令，它

实际上是一个非法的指令。当处理器执行到这个非法指令时，会触发非法指令异常(illegal Instruction Exception)。

在\kern\trap.c文件修改exception_handlers处理异常函数，

代码如下

```
1  case CAUSE_ILLEGAL_INSTRUCTION:
2      // 非法指令异常处理
3      /* LAB1 CHALLENGE3   YOUR CODE :  */
4      /*(1)输出指令异常类型 ( illegal instruction)
5      *(2)输出异常指令地址
6      *(3)更新 tf->epc寄存器
7      */
8      cprintf("Exception type: Illegal instruction\n");
9      cprintf("Ebreak caught at 0x%08x\n", tf->epc);
10     tf->epc += 4;// 更新 tf->epc寄存器
11     break;
12     case CAUSE_BREAKPOINT:
13         //断点异常处理
14         /* LAB1 CHALLENGE3   YOUR CODE :  */
15         /*(1)输出指令异常类型 ( breakpoint)
16         *(2)输出异常指令地址
17         *(3)更新 tf->epc寄存器
18         */
19         cprintf("Exception type: Breakpoint");
20         cprintf("Illegal instruction caught at 0x%08x\n",tf->epc);
21         tf->epc += 4;// 更新 tf->epc寄存器
```

输出结果

```
Special kernel symbols:  
entry 0x000000008020000c (virtual)  
etext 0x0000000080200a6e (virtual)  
edata 0x0000000080204010 (virtual)  
end    0x0000000080204028 (virtual)  
Kernel executable memory footprint: 17KB  
Exception type: BreakpointIllegal instruction caught at 0x80200048
```

```
Special kernel symbols:  
entry 0x000000008020000c (virtual)  
etext 0x0000000080200a6e (virtual)  
edata 0x0000000080204010 (virtual)  
end    0x0000000080204028 (virtual)  
Kernel executable memory footprint: 17KB  
Exception type: Illegal instruction  
Ebreak caught at 0x80200048
```