

# 中断与分时多任务

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

为什么 `while (1);` 死循环不会把电脑卡死？

- 本讲内容
  - 中断机制
  - 实现分时多任务

# 中断机制

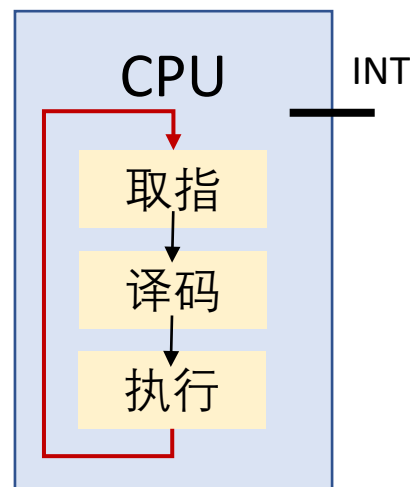
# 回到一个经典问题

为什么 `while (1);` 死循环不会把电脑卡死？

- 因为有中断
  - 相当于在指令之后插入一段代码

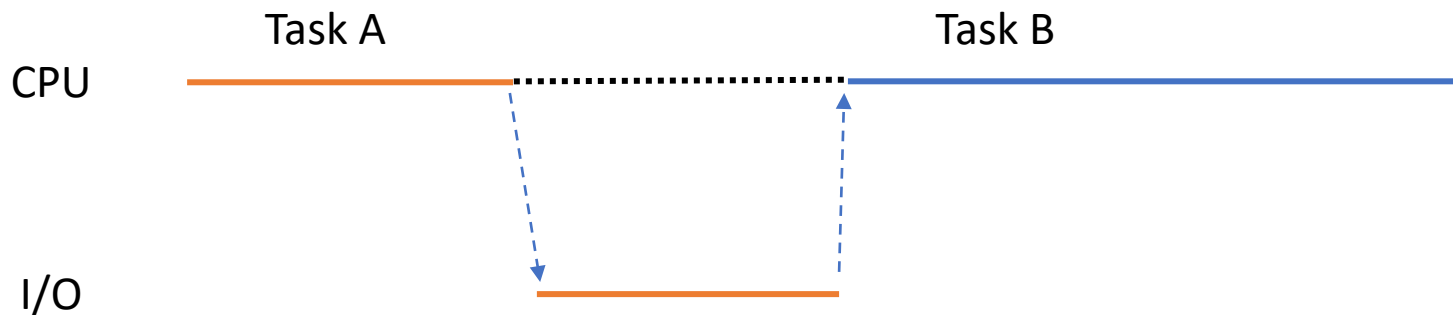
```
if (has_interrupt && int_enabled) {  
    interrupt_handler();  
}
```

- 类似的是异常机制
  - 有时候称为同步中断



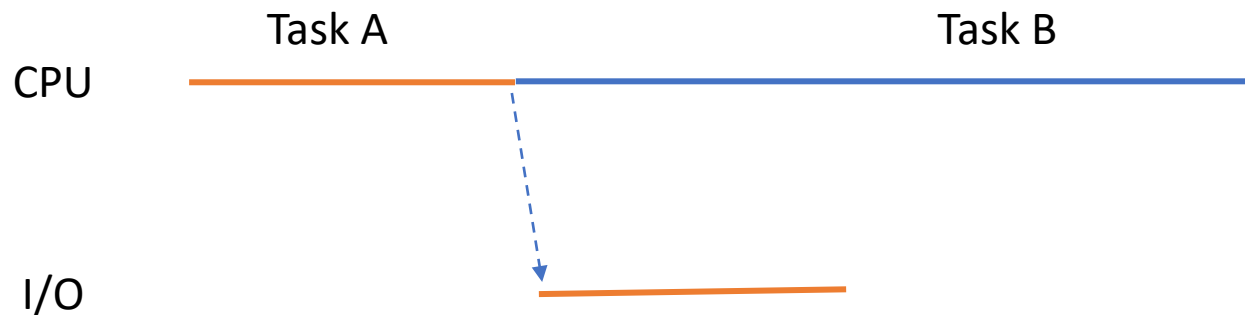
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



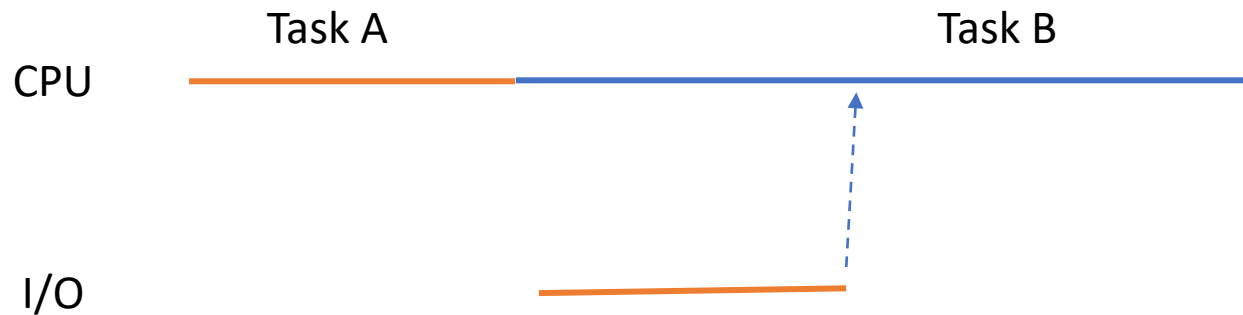
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



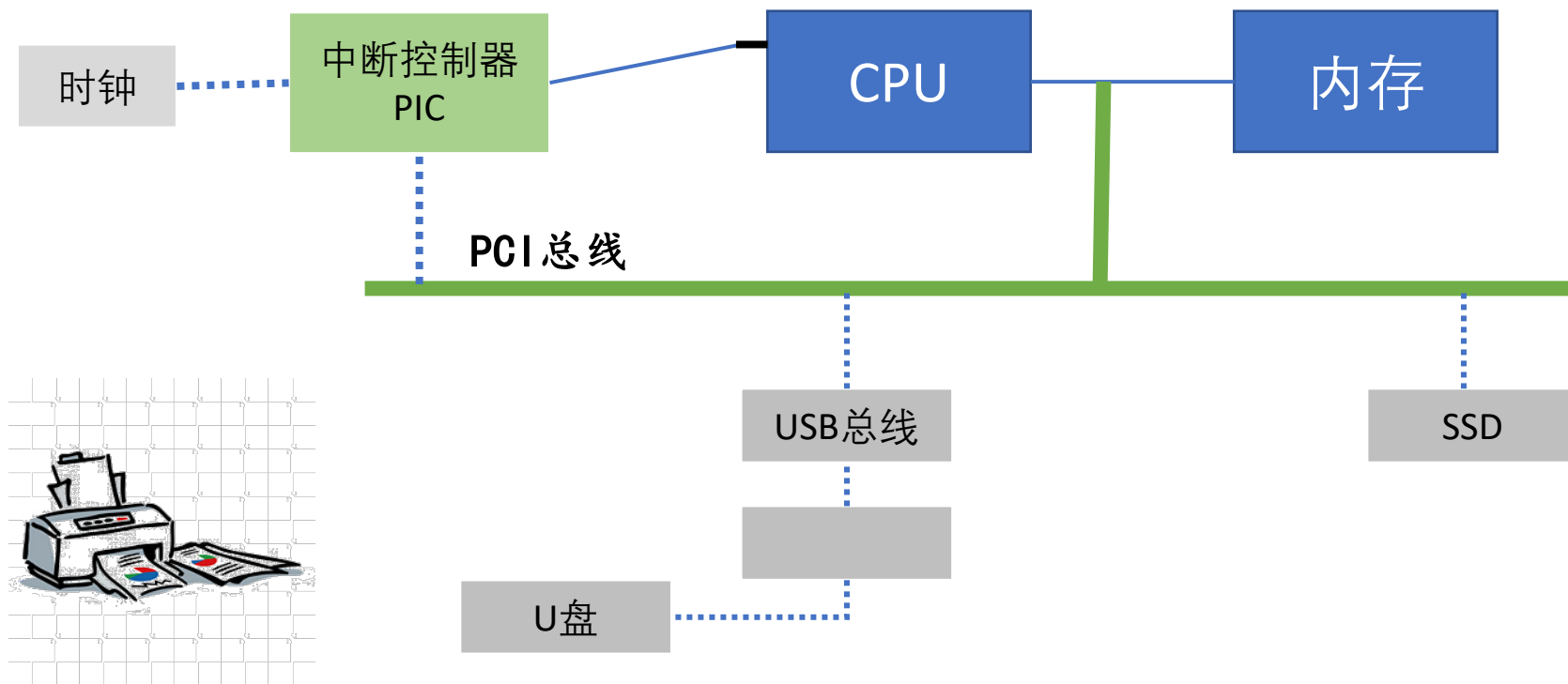
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动的函数调用
  - 相当于在每条语句后都插入

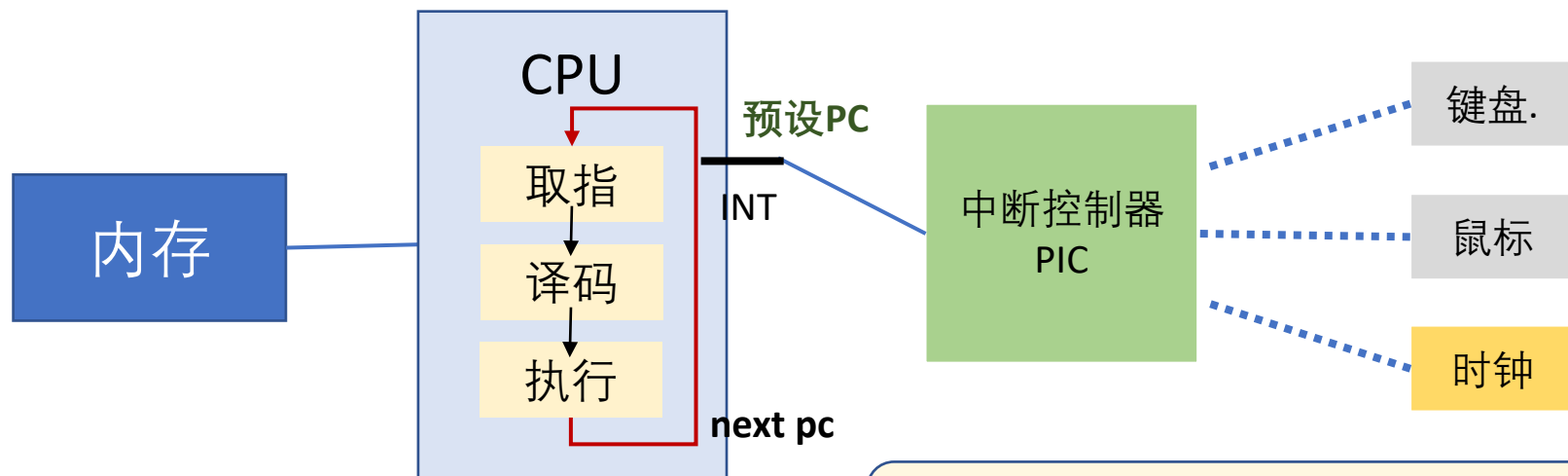
```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
  - 于是就有了最早的操作系统：管理I/O设备的库代码

# I/O设备回顾



# 中断机制总览



- 因为有中断
  - 相当于在指令之后插入一段代码

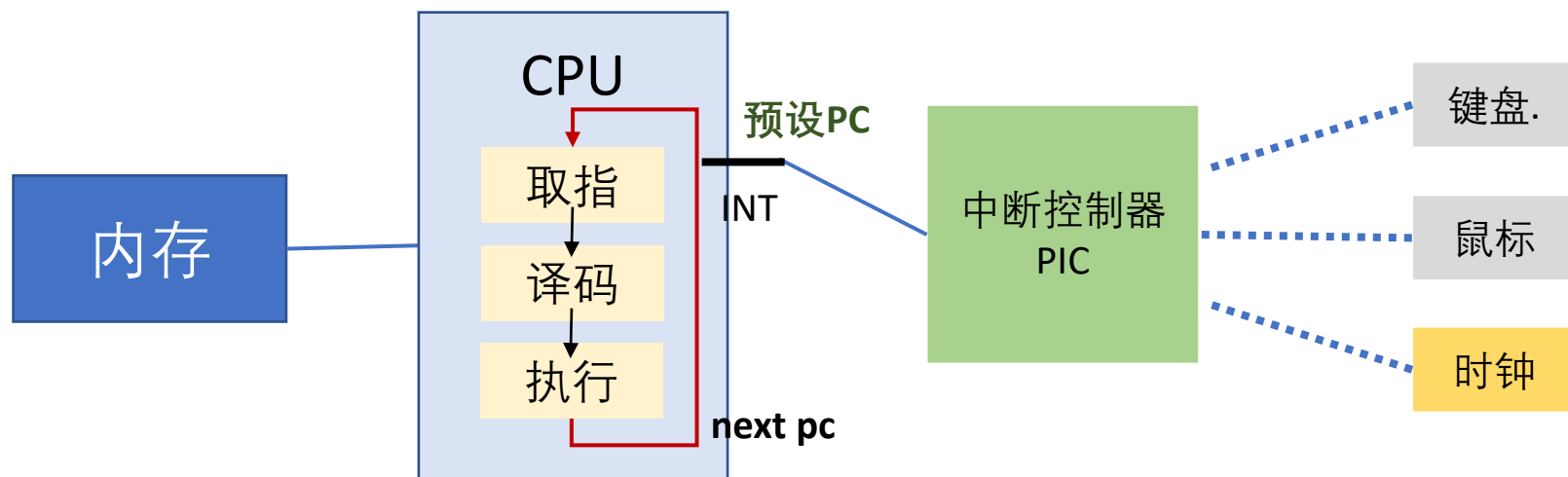
```
if (has_interrupt && int_enabled) {  
    interrupt_handler();  
}
```

Hz

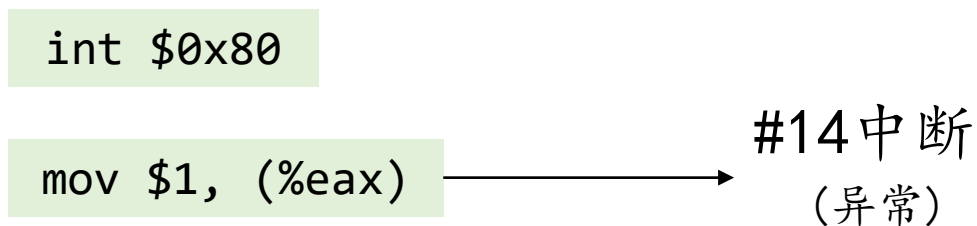


```
counter --;  
if(counter == 0){  
    counter = C;  
    INT = ¬INT;  
}
```

# INT指令和Segmentation fault

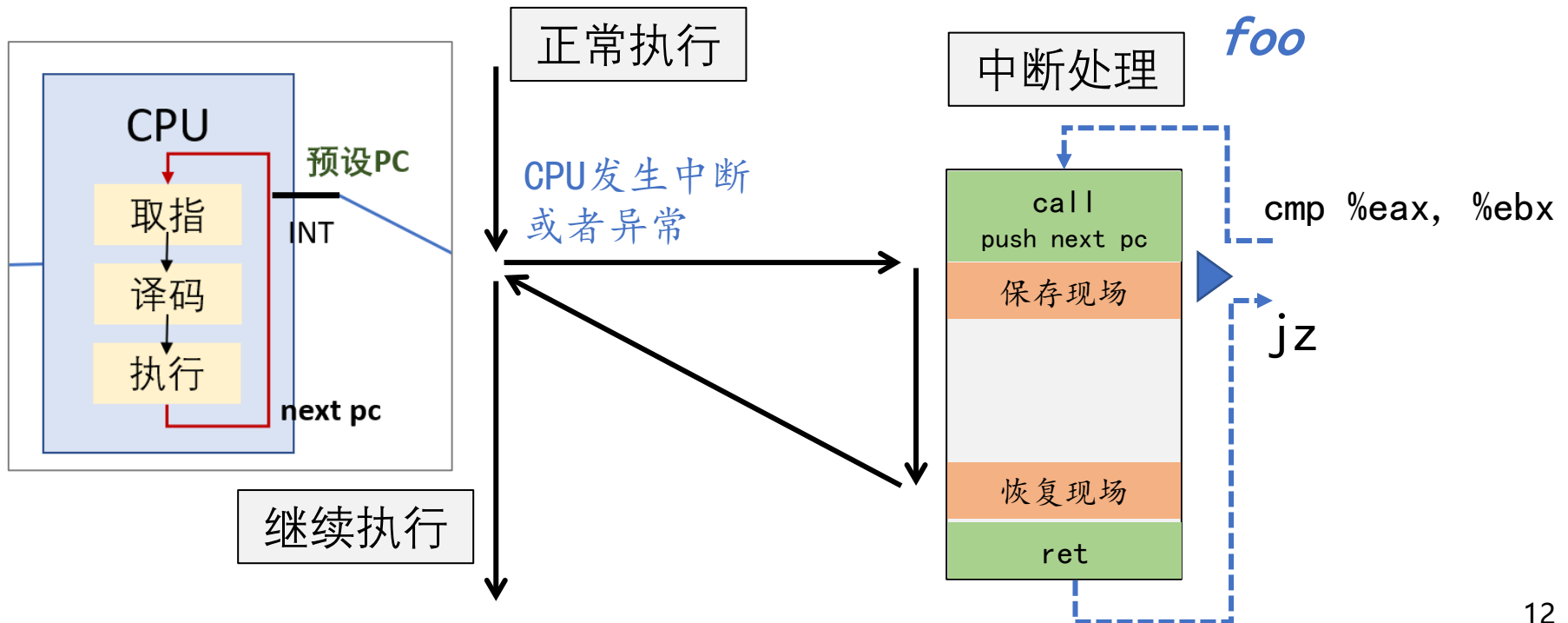


- 类似的是异常机制
  - 有时候称为同步中断



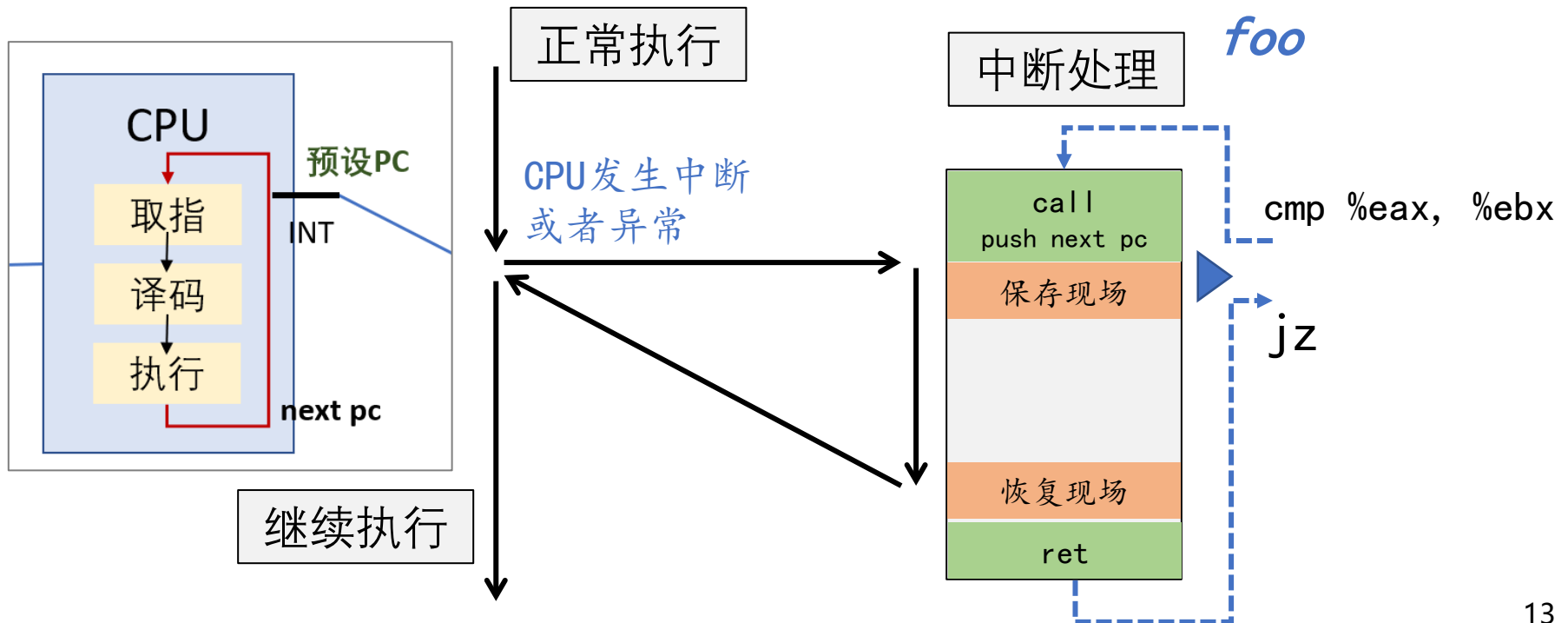
# 中断的实现

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)

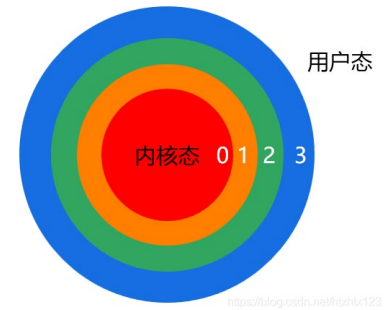


# 中断的实现

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)
  - 但中断不仅要保存 PC (尤其是在有特权级切换的时候)
- 中断处理
  - 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)



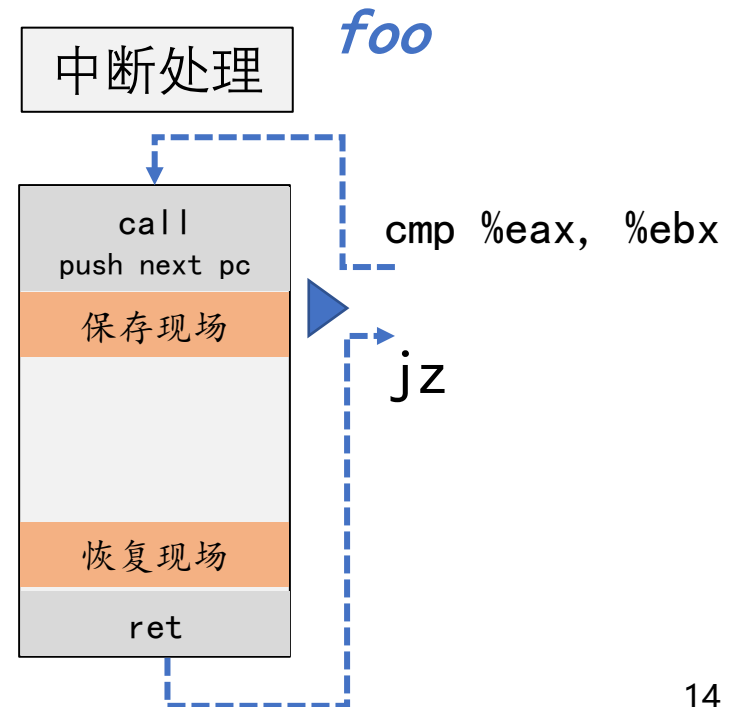
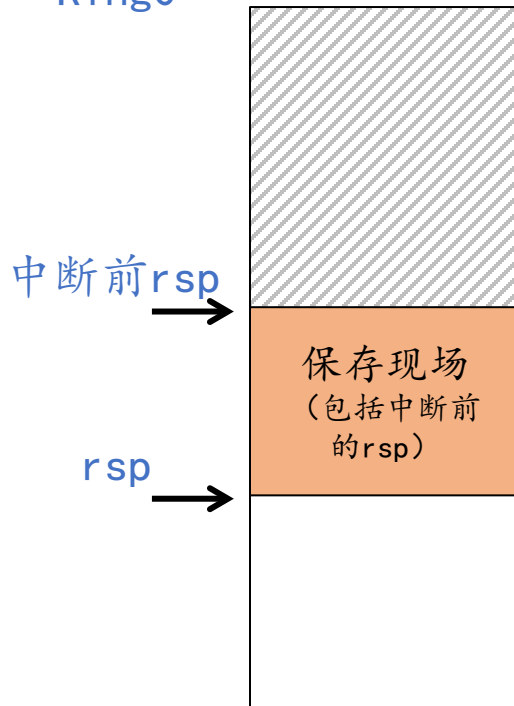
# 中断的实现



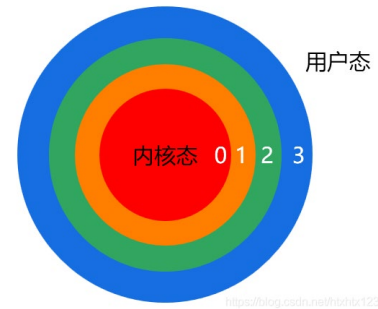
## • 中断处理

- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - `int $0x80` 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断

Ring0



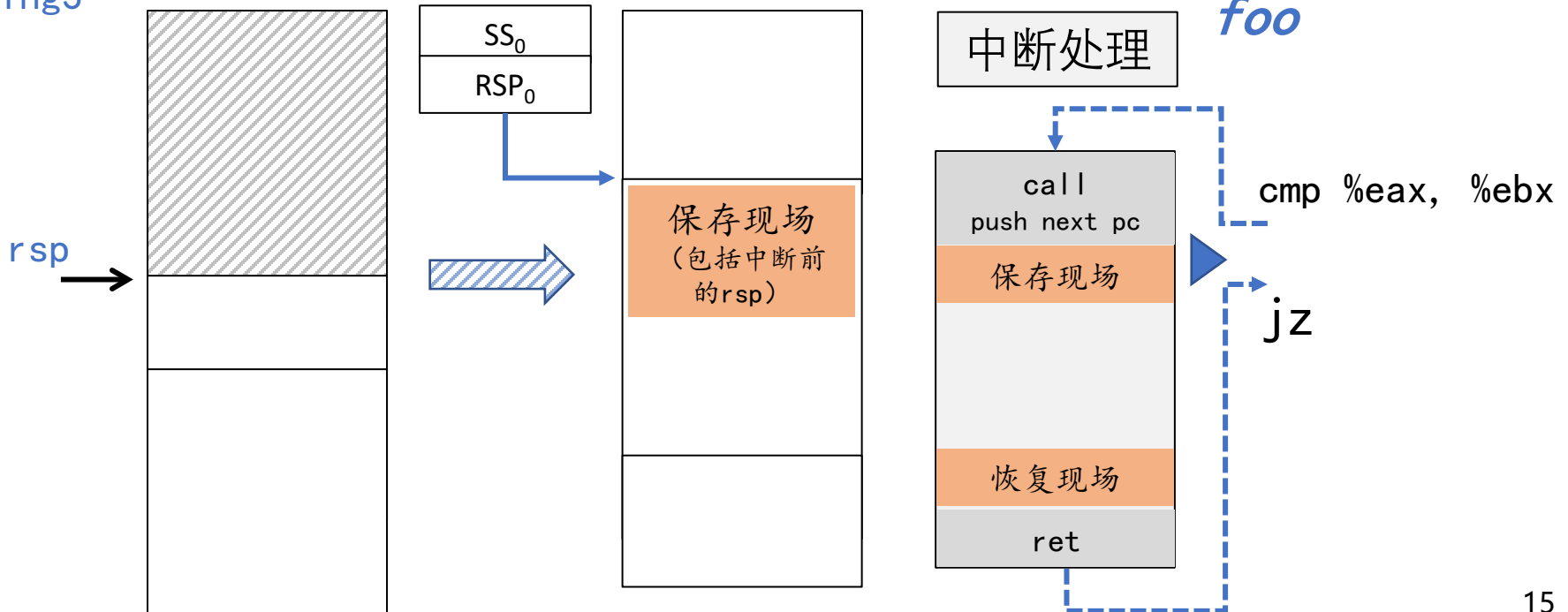
# 中断的实现



## • 中断处理

- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - `int $0x80` 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断

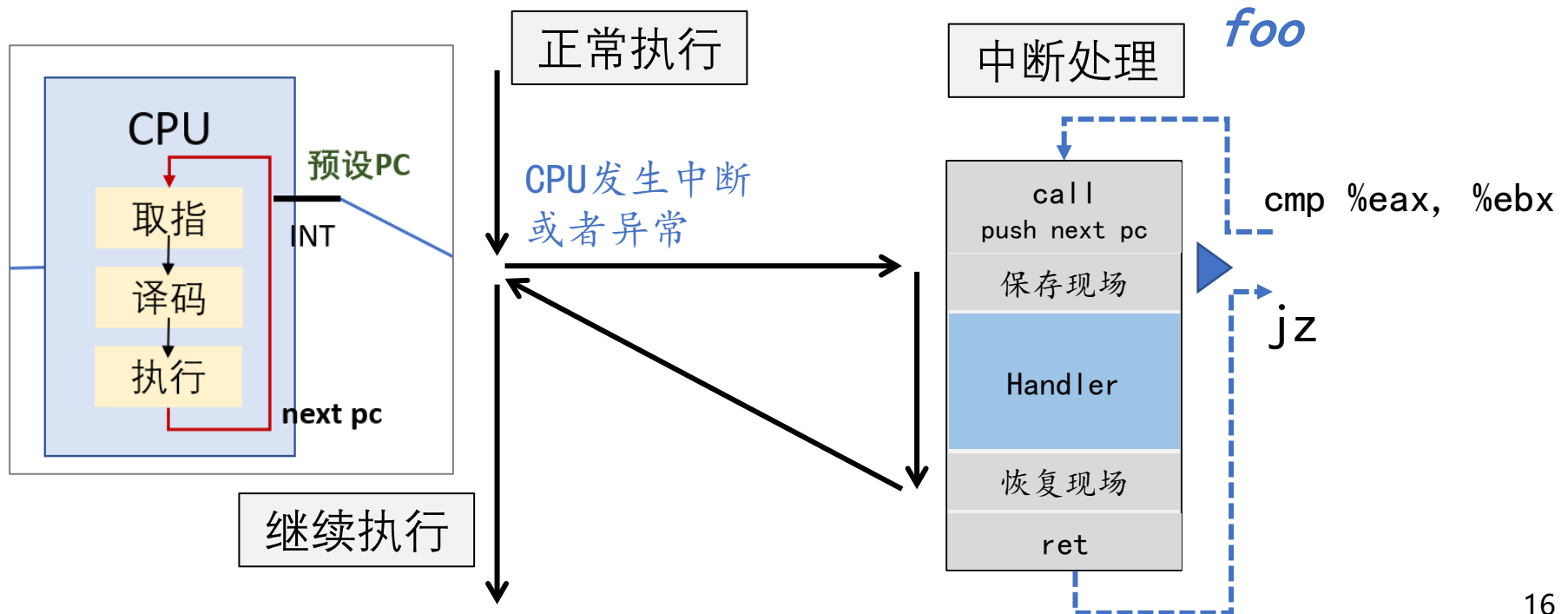
Ring3



# 中断的实现

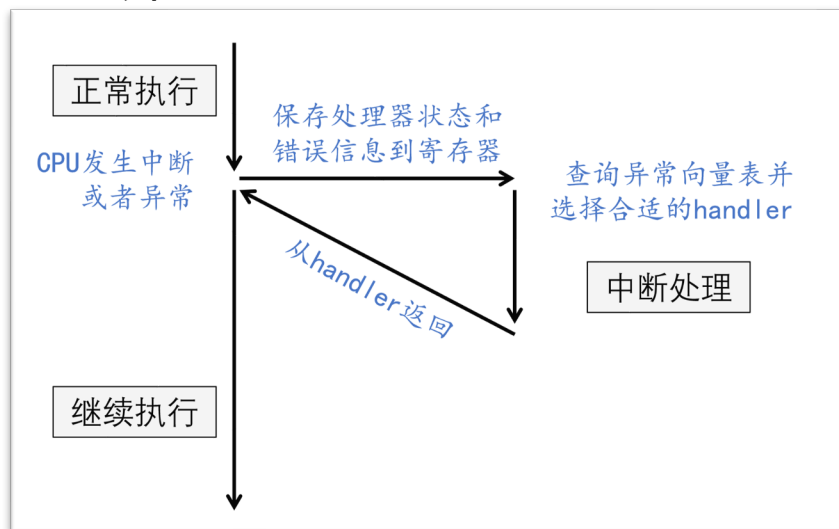
## • 中断处理

- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - `int $0x80` 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断



# x86-64中断过程

- 比 “函数调用” 复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)
  - 但中断不仅要保存 PC (尤其是在有特权级切换的时候)
- 中断处理
  - 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
  - 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
    - int \$0x80 指令可以产生 128 号异常
    - 时钟会产生 32 号中断
    - 键盘会产生 33 号中断
    - .....



# 为什么死循环不会把电脑卡死?

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))

## x86 Instruction Set Reference

### CLI

#### Clear Interrupt Flag

| Opcode | Mnemonic | Description                                                            |
|--------|----------|------------------------------------------------------------------------|
| FA     | CLI      | Clear interrupt flag; interrupts disabled when interrupt flag cleared. |

| Operation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>if(PE == 0) IF = 0; //Reset Interrupt Flag else {     if(VM == 0) {         if(IOPL != CPL) IF = 0; //Reset Interrupt Flag         else {             if(IOPL &lt; CPL &amp;&amp; CPL &lt; 3 &amp;&amp; PVI == 1) VIF = 0; //Reset Virtual Interrupt Flag             else Exception(GP(0));         }     }     else {         if(IOPL == 3) IF = 0; //Reset Interrupt Flag         else {             if(IOPL &lt; 3 &amp;&amp; VME == 1) VIF = 0; //Reset Virtual Interrupt Flag             else Exception(GP(0));         }     } }</pre> |

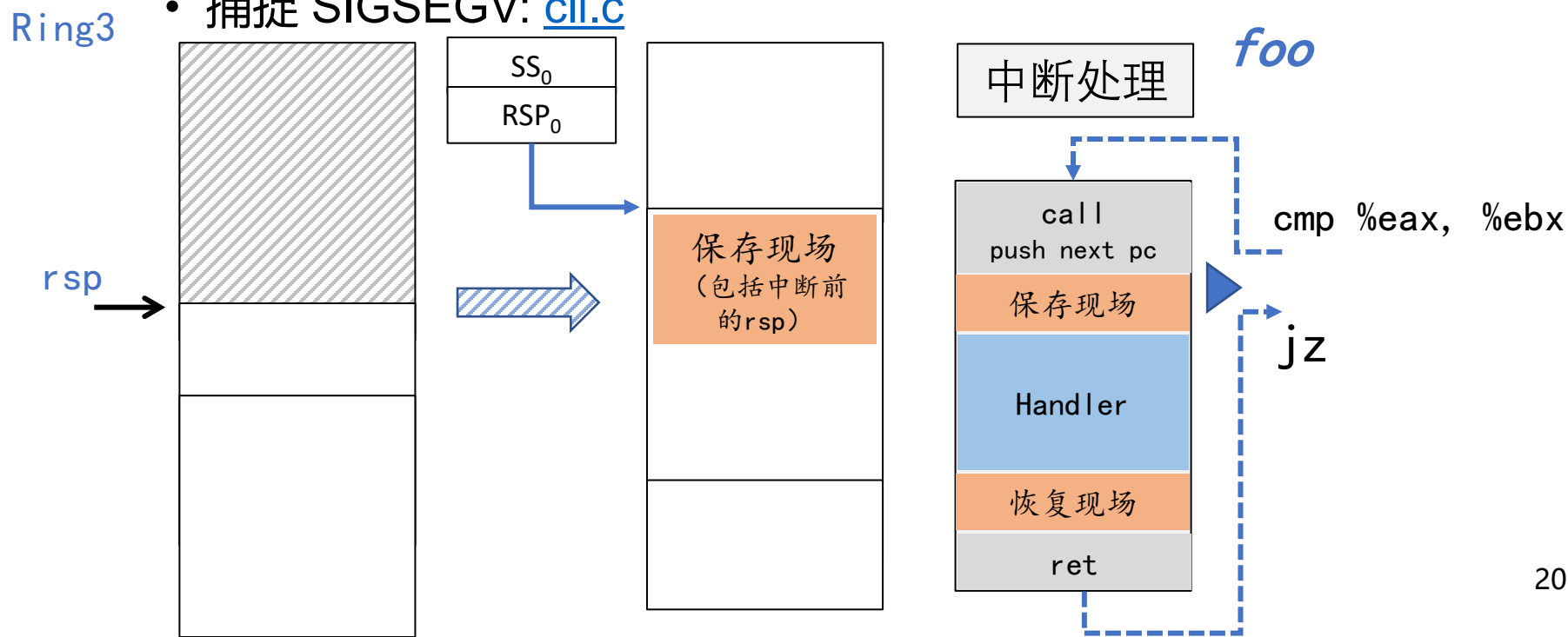
```
1 int main(){  
2     asm volatile ("cli");  
3     while(1);  
4 }
```

~/Documents/ICS2021/teach/Course13/a.c[1] [c] unix utf-8 Ln 1, Col 1/4

# 为什么死循环不会把电脑卡死？

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))
    - 发生 13 号异常
    - Error Code 0
      - 异常机制进入操作系统代码执行
      - 操作系统发送 SIGSEGV 信号

- 捕捉 SIGSEGV: [cli.c](#)





```
1 #define _GNU_SOURCE
2
3 #include <signal.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <ucontext.h>
7 #include <stdint.h>
8
9 void on_sigsegv(int sig, siginfo_t *info, void *ucontext) {
10     ucontext_t *ctx = (ucontext_t *)ucontext;
11     uint8_t *pc = (void *)ctx->uc_mcontext.gregs[REG_RIP];
12     printf("PC = %p ", pc);
13     for (int i = -8; i < 8; i++) {
14         printf(i == 0 ? "[%02x] " : "%02x ", pc[i]);
15     }
16     printf("\n");
17     exit(1);
18 }
19
20 int main() {
21     struct sigaction act;
22     act.sa_sigaction = on_sigsegv;
23     sigemptyset(&act.sa_mask);
24     act.sa_flags = SA_SIGINFO;
25     sigaction(SIGSEGV, &act, NULL);
26     asm volatile("cli");
27 }
```

~/Documents/ICS2021/teach/Course13/cli.c[1]

[c] unix utf-8 Ln 13, Col 2/27

=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')

```
$ ls
a.c  a.out  cli.c  thread-os.c
$ vim cli.c
$ gcc cli.c
$ ./a.out
PC = 0x5634316132ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b
$ █
```

|       |                |            |                      |
|-------|----------------|------------|----------------------|
| 12e3: | bf 0b 00 00 00 | mov        | \$0xb,%edi           |
| 12e8: | e8 c3 fd ff ff | call       | 10b0 <sigaction@plt> |
| 12ed: | fa             | <b>cli</b> |                      |
| 12ee: | b8 00 00 00 00 | mov        | \$0x0,%eax           |
| 12f3: | 48 8b 4d f8    | mov        | -0x8(%rbp),%rcx      |

```
$ ls
a.c  a.out  cli.c  thread-os.c
$ vim cli.c
$ gcc cli.c
$ ./a.out
PC = 0x5634316132ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b
$
```

```
$ ./a.out
```

```
PC = 0x5620206812ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b
```

```
$ ./a.out
```

```
PC = 0x555cd8d502ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b
```

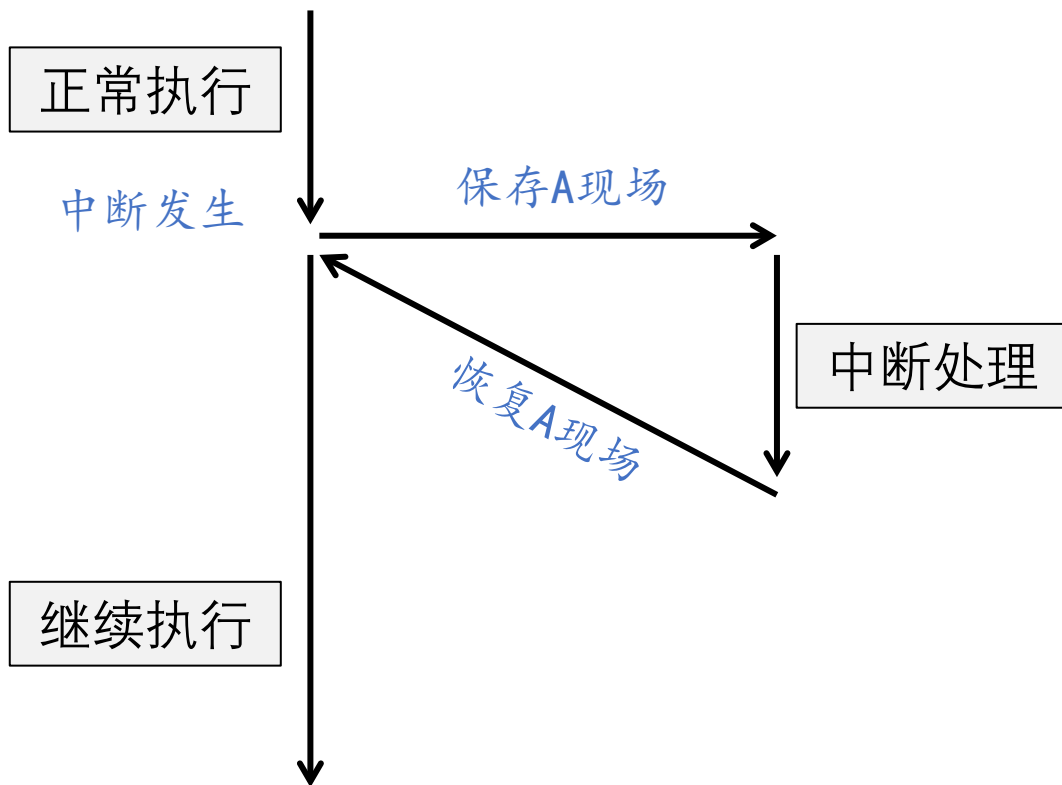
```
$ █
```

# 为什么死循环不会把电脑卡死？

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))
    - 发生 13 号异常
    - Error Code 0
      - 异常机制进入操作系统代码执行
      - 操作系统发送 SIGSEGV 信号
  - 捕捉 SIGSEGV: [cli.c](#)
- 中断发生后，操作系统代码将会切换到另一个进程执行
  - “调度”：让进程公平地共享 CPU

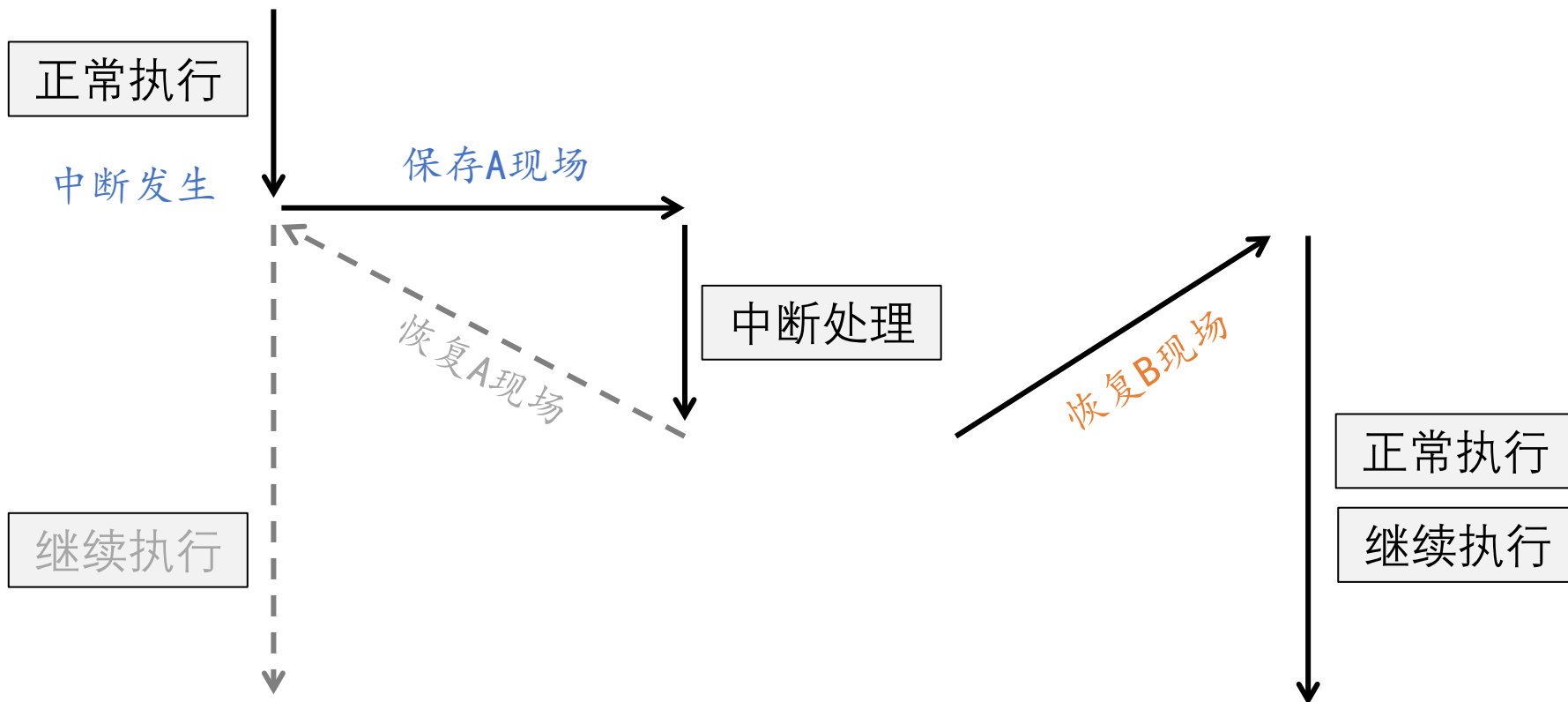
# 实现分时多任务

A



A

B



# 代码选讲

---

- Abstract Machine
  - trap64.S
  - cte.c
- 调试 “迷你” 操作系统
  - [thread-os.c](#)
    - -s -S 启动 QEMU
    - gdb target remote localhost:1234
      - [gdbnotes for CSAPP](#)

```

1 #include <am.h>
2 #include <klib.h>
3 #include <klib-macros.h>
4
5 typedef union task {
6     struct {
7         const char *name;
8         union task *next;
9         void (*entry)(void *);
10        Context *context;
11    };
12    uint8_t stack[8192];
13 } Task;
14
15 Task *current;
16
17 void func(void *arg) {
18     while (1) {
19         putchar(*(char *)arg);
20         for (int volatile i = 0; i < 100000; i++) ;
21     }
22 }
23
24 Task tasks[] = {
25     { .name = "a", .entry = func },
26     { .name = "b", .entry = func },
27     { .name = "c", .entry = func },

```

```

24 Task tasks[] = {
25     { .name = "a", .entry = func },
26     { .name = "b", .entry = func },
27     { .name = "c", .entry = func },
28 };
29
30 Context *on_interrupt(Event ev, Context *ctx) {
31     if (!current) current = &tasks[0];
32     else          current->context = ctx;
33     current = current->next;
34     return current->context;
35 }
36
37 int main() {
38     ioe_init();
39     cte_init(on_interrupt);
40
41     for (int i = 0; i < LENGTH(tasks); i++) {
42         Task *task      = &tasks[i];
43         Area stack      = (Area) { &task->context + 1, task + 1 };
44         task->context = kcontext(stack, task->entry, (void *)task->name);
45         task->next    = &tasks[(i + 1) % LENGTH(tasks)];
46     }
47     iset(true);
48     while(1);
49 }

```

<s2021/am-kernels/kernels/thread-os/thread-os.c[1] [c] unix utf-8 Ln 49, Col 1/49

\$ ■

```
$ ls  
build Makefile thread-os.c  
$
```

```
$ ls  
thread-os-x86_64-qemu  thread-os-x86_64-qemu.elf  x86_64-qemu  
$
```

```
—/home/why/Documents/ICS2021/ics2021/am-kernels/kernels/thread-os/thread-os.c—
```

```
30     Context *on_interrupt(Event ev, Context *ctx) {
31         if (!current) current = &tasks[0];
32         else             current->context = ctx;
33         current = current->next;
34         return current->context;
35     }
36
B+ 37     int main() {
38         ioe_init();
>39         cte_init(on_interrupt);
40
41         for (int i = 0; i < LENGTH(tasks); i++) {
42             Task *task      = &tasks[i];
43             Area stack      = (Area) { &task->context + 1, task + 1 };
44             task->context = kcontext(stack, task->entry, (void *)task->na
45             task->next     = &tasks[(i + 1) % LENGTH(tasks)];
```

```
remote Thread 1.1 In: main
```

```
L39 PC: 0x1000d3
```

```
(gdb) n
```

```
(gdb) █
```

```

0x1016e0 <iset>                                endbr64
0x1016e4 <iset+4>                              test    %dil,%dil
0x1016e7 <iset+7>                              je      0x1016f0 <iset+16>
0x1016e9 <iset+9>                              sti
>0x1016ea <iset+10>                            ret
0x1016eb <iset+11>                            nopl    0x0(%rax,%rax,1)
0x1016f0 <iset+16>                            cli
0x1016f1 <iset+17>                            ret
0x1016f2                                     data16 nopw %cs:0x0(%rax,%rax,1)
0x1016fd                                     nopl    (%rax)
0x101700 <__am_panic_on_return>                endbr64
0x101704 <__am_panic_on_return+4>              push    %rbx
0x101705 <__am_panic_on_return+5>              mov     $0x41,%edi
0x10170a <__am_panic_on_return+10>             lea     0x1239(%rip),%rbx          # 0x
0x101711 <__am_panic_on_return+17>             nopl    0x0(%rax)
0x101718 <__am_panic_on_return+24>            call    0x1002b0 <putch>

```

remote Thread 1.1 In: iset L?? PC: 0x1016ea

```

rip      0x1016ea      0x1016ea <iset+10>
eflags   0x202        [ IOPL=0 IF ]
cs       0x8          8
ss       0x0          0
ds       0x0          0
es       0x0          0
fs       0x0          0
gs       0x0          0

```

--Type <RET> for more, q to quit, c to continue without paging--

```

0x100132 <main+130>      sub    %rdx,%rax
0x100135 <main+133>      shl    $0xd,%rax
0x100139 <main+137>      add    %rbp,%rax
0x10013c <main+140>      mov    %rax,-0x3ff8(%rbx)
0x100143 <main+147>      cmp    $0x4,%r13
0x100147 <main+151>      jne    0x1000ed <main+61>
0x100149 <main+153>      mov    $0x1,%edi
0x10014e <main+158>      call  0x1016e0 <iset>
>0x100153 <main+163>      jmp    0x100153 <main+163>
0x100155                nopw    %cs:0x0(%rax,%rax,1)
0x10015f                nop
0x100160 <on_interrupt>      endbr64
0x100164 <on_interrupt+4>  mov    $0x109700,%rdx
0x10016b <on_interrupt+11> mov    (%rdx),%rax
0x10016e <on_interrupt+14> test   %rax,%rax
0x100171 <on_interrupt+17> je     0x100188 <on_interrupt+40>

```

remote Thread 1.1 In: main L48 PC: 0x100153

```

cr2      0x0      0
cr3      0x1000    [ PDBR=0 PCID=0 ]
cr4      0x20     [ PAE ]
cr8      0x0      0

```

--Type <RET> for more, q to quit, c to continue without paging--q

Quit

(gdb) si

(gdb) si

(gdb)

## Useful information

|                   |                                                                |
|-------------------|----------------------------------------------------------------|
| backtrace         | Print the current address and stack backtrace                  |
| where             | Print the current address and stack backtrace                  |
| info program      | Print current status of the program)                           |
| info functions    | Print functions in program                                     |
| info stack        | Print backtrace of the stack)                                  |
| info frame        | Print information about the current stack frame                |
| info registers    | Print registers and their contents                             |
| info breakpoints  | Print status of user-settable breakpoints                      |
| display /FMT EXPR | Print expression EXPR using format FMT<br>every time GDB stops |
| undisplay         | Turn off display mode                                          |
| help              | Get information about gdb                                      |

|                |                                                                              |
|----------------|------------------------------------------------------------------------------|
| x/2w \$rsp     | Examine two (4-byte) words starting at address<br>in \$rsp                   |
| x/2wd \$rsp    | Examine two (4-byte) words starting at address<br>in \$rsp. Print in decimal |
| x/g \$rsp      | Examine (8-byte) word starting at address in \$rsp.                          |
| x/gd \$rsp     | Examine (8-byte) word starting at address in \$rsp.<br>Print in decimal      |
| x/a \$rsp      | Examine address in \$rsp. Print as offset from<br>previous global symbol.    |
| x/s 0xbffff890 | Examine a string stored at 0xbffff890                                        |
| x/20b sum      | Examine first 20 opcode bytes of function sum                                |
| x/10i sum      | Examine first 10 instructions of function sum                                |

```

0x100118 <main+104>    mov    %r13,%rax
0x10011b <main+107>    mul    %r12
0x10011e <main+110>    mov    %rdx,%rax
0x100121 <main+113>    and    $0xfffffffffffffffe,%rdx
0x100125 <main+117>    shr    %rax
0x100128 <main+120>    add    %rax,%rdx
0x10012b <main+123>    mov    %r13,%rax
0x10012e <main+126>    add    $0x1,%r13
0x100132 <main+130>    sub    %rdx,%rax
0x100135 <main+133>    shl    $0xd,%rax
0x100139 <main+137>    add    %rbp,%rax
0x10013c <main+140>    mov    %rax,-0x3ff8(%rbx)
0x100143 <main+147>    cmp    $0x4,%r13
0x100147 <main+151>    jne    0x1000ed <main+61>
0x100149 <main+153>    mov    $0x1,%edi
0x10014e <main+158>    call   0x1016e0 <iset>

```

remote Thread 1.1 In: am\_irq32 L?? PC: 0x102855

```

rip      0x102855      0x102855 <__am_irq32>
eflags   0x202        [ IOPL=0 IF ]
cs        0x8          8
ss        0x0          0
ds        0x0          0
es        0x0          0
fs        0x0          0
gs        0x0          0

```

--Type <RET> for more, q to quit, c to continue without paging--

```

0x100118 <main+104>    mov    %r13,%rax
0x10011b <main+107>    mul    %r12
0x10011e <main+110>    mov    %rdx,%rax
0x100121 <main+113>    and    $0xfffffffffffffffe,%rdx
0x100125 <main+117>    shr    %rax
0x100128 <main+120>    add    %rax,%rdx
0x10012b <main+123>    mov    %r13,%rax
0x10012e <main+126>    add    $0x1,%r13
0x100132 <main+130>    sub    %rdx,%rax
0x100135 <main+133>    shl    $0xd,%rax
0x100139 <main+137>    add    %rbp,%rax
0x10013c <main+140>    mov    %rax,-0x3ff8(%rbx)
0x100143 <main+147>    cmp    $0x4,%r13
0x100147 <main+151>    jne    0x1000ed <main+61>
0x100149 <main+153>    mov    $0x1,%edi
0x10014e <main+158>    call   0x1016e0 <iset>

```

remote Thread 1.1 In: \_\_am\_irq32 L?? PC: 0x102855

fs 0x0 0

gs 0x0 0

--Type <RET> for more, q to quit, c to continue without paging--q

Quit

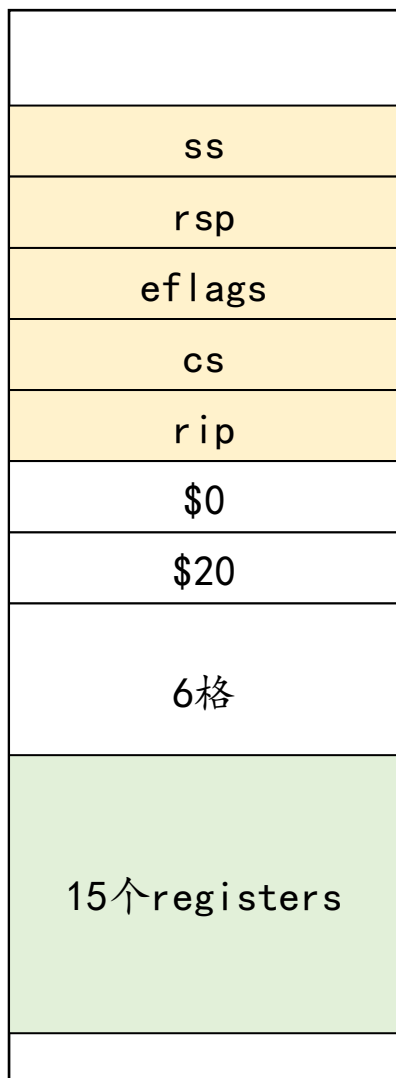
(gdb) x/5gx \$rsp

0x20c848 <\_\_am\_cpuinfo+8264>: 0x00000000000100153 0x0000000000000008

0x20c858 <\_\_am\_cpuinfo+8280>: 0x00000000000000202 0x0000000000020c870

0x20c868 <\_\_am\_cpuinfo+8296>: 0x00000000000000000

(gdb) █



rsp = 0x20c870      ss = 0x0  
rip = 0x100153      cs = 0x8  
eflags = 0x202

rsp = 0x20c848      ss = 0x0  
rip = 0x102855      cs = 0x8  
eflags = 0x202

```

void __am_irq_handle(struct trap_frame *tf) {
    Context *saved_ctx = &tf->savd_context;
    Event ev = {
        .event = EVENT_NULL,
        .cause = 0, .ref = 0,
        .msg = "(no message)",
    };

#ifdef __x86_64
    saved_ctx->rip      = tf->rip;
    saved_ctx->cs       = tf->cs;
    saved_ctx->rflags   = tf->rflags;
    saved_ctx->rsp      = tf->rsp;
    saved_ctx->rsp0     = CPU->tss.rsp0;
    saved_ctx->ss       = tf->ss;
#else
    saved_ctx->eip      = tf->eip;
    saved_ctx->cs       = tf->cs;
    saved_ctx->eflags   = tf->eflags;
    saved_ctx->esp0     = CPU->tss.esp0;
    saved_ctx->ss3      = USEL(SEG_UDATA);
    // no ss/esp saved for DPL_KERNEL
    saved_ctx->esp = (tf->cs & DPL_USER ? tf->esp : (uint32_t)(tf + 1) - 8);
#endif
    saved_ctx->cr3      = (void *)get_cr3();
}

```

```

0x102770 <trap+13>      push    %r11
0x102772 <trap+15>      push    %r10
0x102774 <trap+17>      push    %r9
0x102776 <trap+19>      push    %r8
0x102778 <trap+21>      push    %rdi
0x102779 <trap+22>      push    %rsi
0x10277a <trap+23>      push    %rbp
0x10277b <trap+24>      push    %rdx
0x10277c <trap+25>      push    %rcx
0x10277d <trap+26>      push    %rbx
0x10277e <trap+27>      push    %rax
0x10277f <trap+28>      push    $0x0
0x102781 <trap+30>      mov     %rsp,%rdi
>0x102784 <trap+33>      call    0x100750 <__am_irq_handle>
0x102789 <__am_iret>      mov     %rdi,%rsp
0x10278c <__am_iret+3>     mov     0xa0(%rsp),%rax

```

remote Thread 1.1 In: trap

L?? PC: 0x102784

```

0x00000000000010277a in trap ()
0x00000000000010277b in trap ()
0x00000000000010277c in trap ()
0x00000000000010277d in trap ()
0x00000000000010277e in trap ()
0x00000000000010277f in trap ()
0x000000000000102781 in trap ()
0x000000000000102784 in trap ()

```

(gdb) █

```

> 0x102789 < __am_iret> mov %rdi,%rsp
0x10278c < __am_iret+3> mov 0xa0(%rsp),%rax
0x102794 < __am_iret+11> mov %eax,%ds
0x102796 < __am_iret+13> mov %eax,%es
0x102798 < __am_iret+15> add $0x8,%rsp
0x10279c < __am_iret+19> pop %rax
0x10279d < __am_iret+20> pop %rbx
0x10279e < __am_iret+21> pop %rcx
0x10279f < __am_iret+22> pop %rdx
0x1027a0 < __am_iret+23> pop %rbp
0x1027a1 < __am_iret+24> pop %rsi
0x1027a2 < __am_iret+25> pop %rdi
0x1027a3 < __am_iret+26> pop %r8
0x1027a5 < __am_iret+28> pop %r9
0x1027a7 < __am_iret+30> pop %r10
0x1027a9 < __am_iret+32> pop %r11

```

remote Thread 1.1 In: \_\_am\_iret L?? PC: 0x102789

```

0x00000000001008a0 in __am_irq_handle ()
0x00000000001008a4 in __am_irq_handle ()
0x00000000001008a7 in __am_irq_handle ()
0x00000000001008a8 in __am_irq_handle ()
0x00000000001008a9 in __am_irq_handle ()
0x0000000000102789 in __am_iret ()

```

(gdb) █

```

0x1001a8 <func+8>      sub    $0x10,%rsp
0x1001ac <func+12>     nopl    0x0(%rax)
0x1001b0 <func+16>     movsbl (%rbx),%edi
0x1001b3 <func+19>     call   0x1002b0 <putch>
0x1001b8 <func+24>     movl    $0x0,0xc(%rsp)
0x1001c0 <func+32>     mov     0xc(%rsp),%eax
0x1001c4 <func+36>     cmp     $0x1869f,%eax
0x1001c9 <func+41>     jg      0x1001b0 <func+16>
>0x1001cb <func+43>     mov     0xc(%rsp),%eax
0x1001cf <func+47>     add     $0x1,%eax
0x1001d2 <func+50>     mov     %eax,0xc(%rsp)
0x1001d6 <func+54>     mov     0xc(%rsp),%eax
0x1001da <func+58>     cmp     $0x1869f,%eax
0x1001df <func+63>     jle     0x1001cb <func+43>
0x1001e1 <func+65>     jmp     0x1001b0 <func+16>
0x1001e3                nopw    %cs:0x0(%rax,%rax,1)

```

remote Thread 1.1 In: func

L20

PC: 0x1001cb

(gdb) x/5gx \$rsp

```

0x107258 <tasks+16248>: 0x000000000001001cb      0x00000000000000008
0x107268 <tasks+16264>: 0x00000000000000297      0x000000000001072c0
0x107278 <tasks+16280>: 0x00000000000000000

```

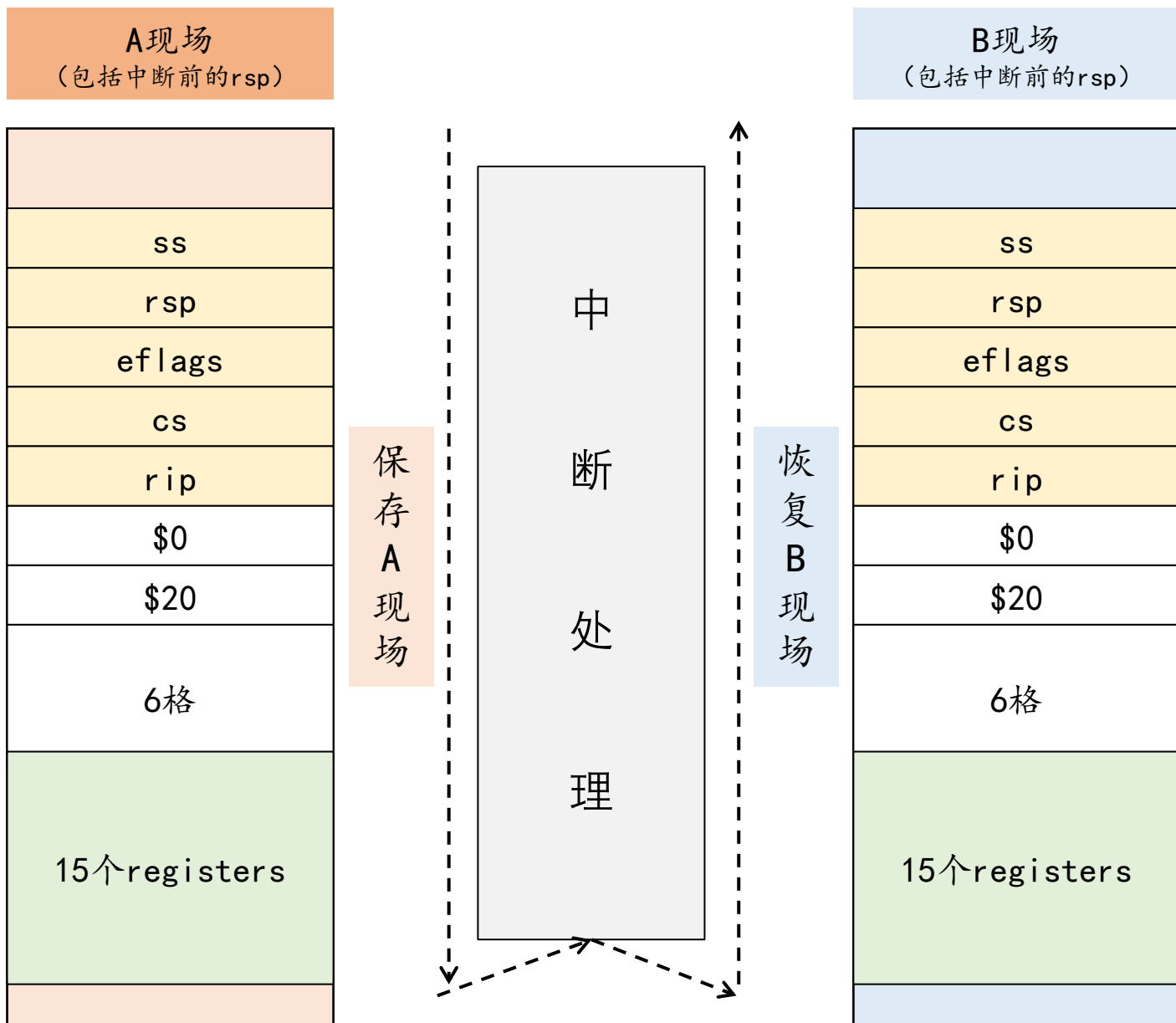
(gdb) si

0x000000000001001cb in func (arg=0x102922)

at /home/why/Documents/ICS2021/ics2021/am-kernels/kernels/thread-os/thread-os.

c:20

(gdb)



# 中断 + 更大的内存 = 分时多线程

```
void foo() { while (1) printf("a"); }  
void bar() { while (1) printf("b"); }
```

- 能够让foo()和bar() “同时” 在处理器上执行?
  - 借助每条语句后被动插入的interrupt\_handler()调用

```
void interrupt_handler() {  
    dump_regs(current->regs);  
    current = (current->func == foo) ? bar : foo;  
    restore_regs(current->regs);  
}
```

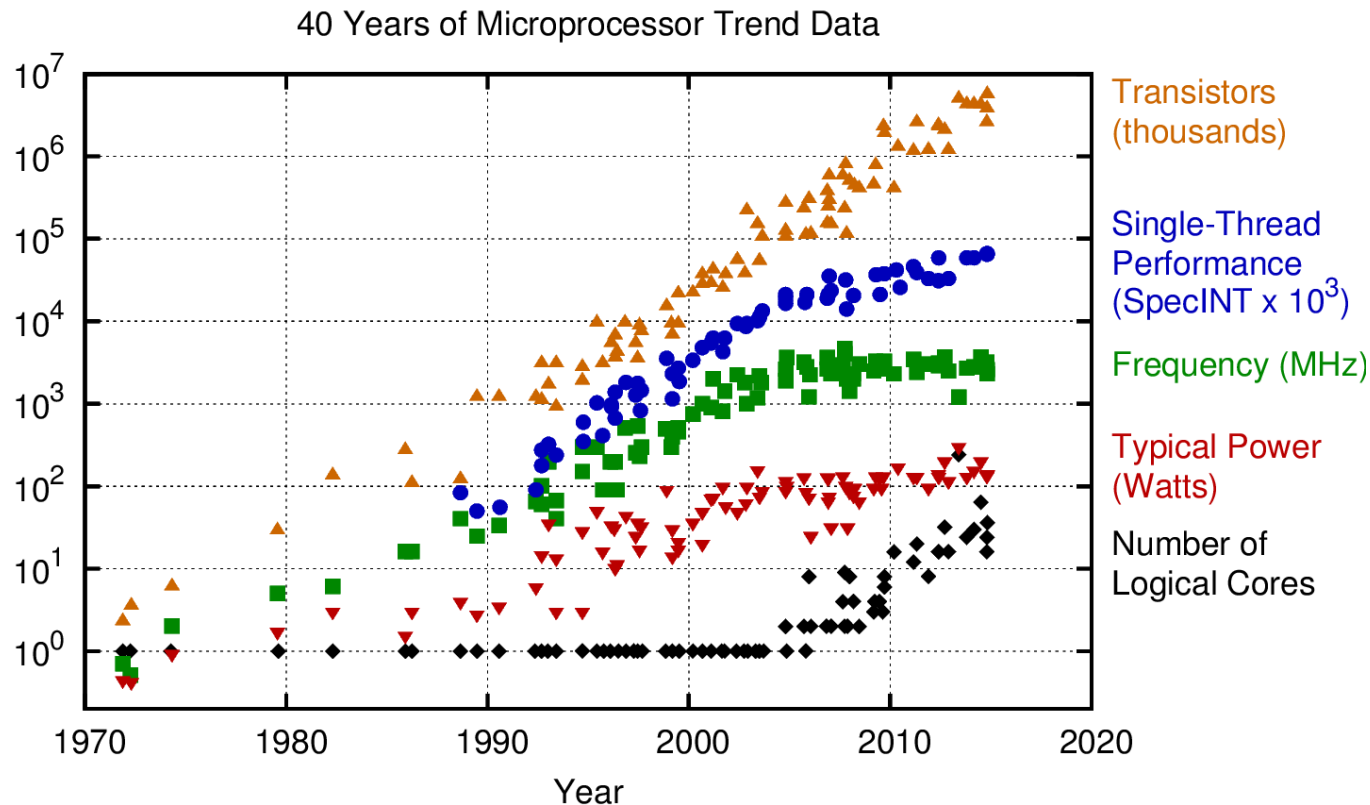
- 操作系统背负了 “调度” 的职责

# 分时多线程 + 虚拟存储 = 进程

- 让foo()和bar()的执行互相不受影响
  - 在计算机系统里增加映射函数 $f_{\text{foo}}, f_{\text{bar}}$
  - foo访问内存地址 $m$ 时, 将被重定位到 $f_{\text{foo}}(m)$
  - bar访问内存地址 $m$ 时, 将被重定位到 $f_{\text{bar}}(m)$
- foo和bar本身无权管理 $f$
- 操作系统需要完成进程、存储、文件的管理
- UNIX诞生 (就是我们今天的进程; Android app; ...)
  - `gcc a.c`
  - `readelf -a a.out` → 二进制文件的全部信息

# 故事其实没有停止.

- 面对有限的功耗、难以改进的制程、无法提升的频率、应用需求
  - 多处理器、big.LITTLE、异构处理器 (GPU、NPU、...)
  - 单指令多数据(MMX, SSE, AVX, ...), 虚拟化(VT; EL0/1/2/3), ... , 安全执行环节 (TrustZone; SGX), ...



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

# 总结

# 操作系统：中断驱动的上下文切换

---

- 应用程序视角

- 一组系统调用的 API
  - 进程管理、存储管理、文件管理.....

- 硬件视角

- 操作系统是一个中断处理程序
  - 设备中断：上下文切换；调用设备驱动程序； .....
  - 系统调用：操作系统代码执行 (API 实现)
  - 异常：发送信号 (类似系统调用)

End.