第14讲 多处理器系统和中断机制

2024.04.15

中断机制：奠定操作系统的霸主地位。

1. 多处理器和中断

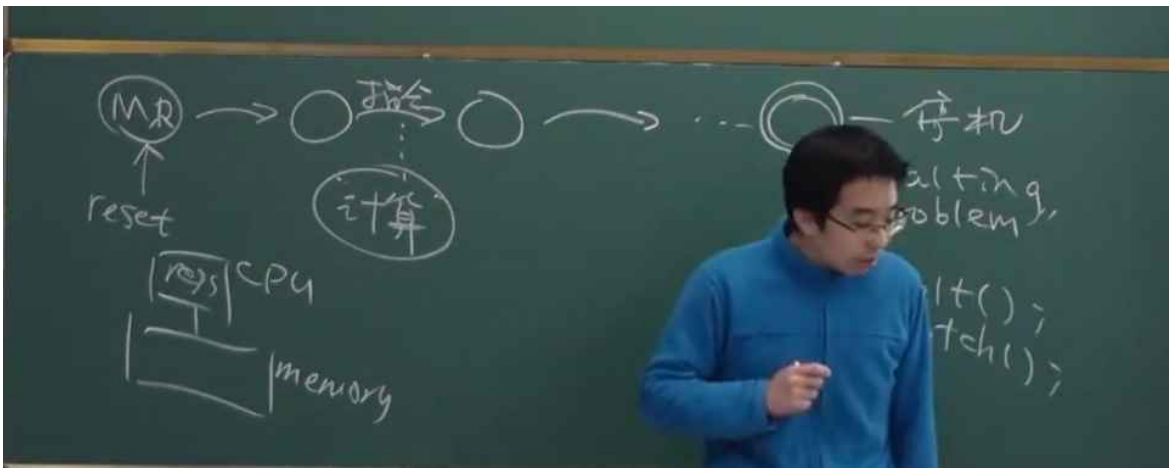
比如说如果我们要实现一个线程库，那么这个线程库到底是怎么在现代计算机系统上实现的？如果我们的计算机系统上只有一个CPU，那么这一个 CPU 是怎么样实现我们的多个线程呢？

1. 多处理器

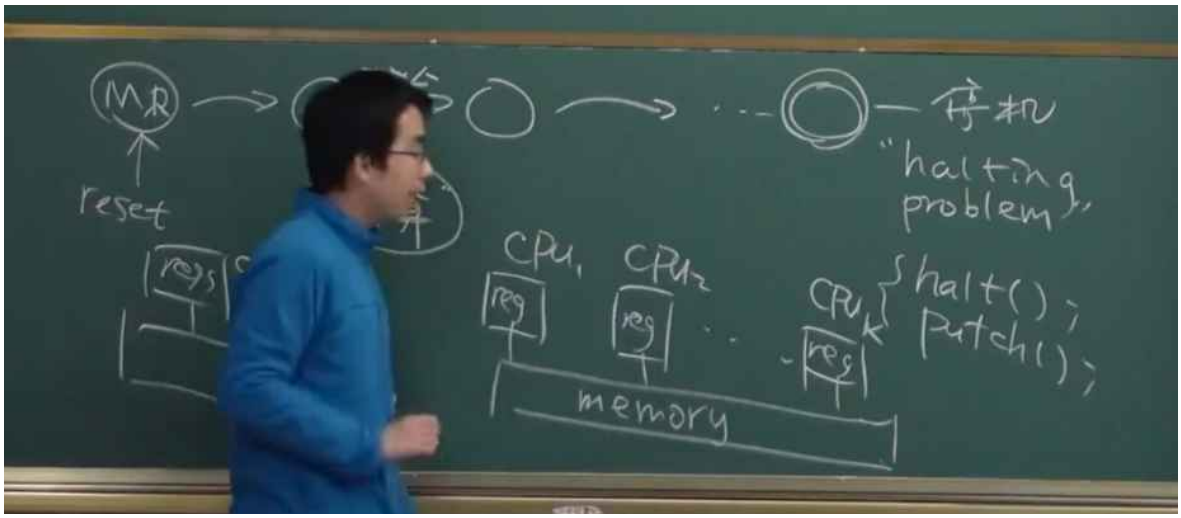
从图灵机到计算机系统：

然后我们从一个单处理器到了多处理器：每个系统的处理器就相当于一个线程。

我们程序的状态是内存和寄存器。

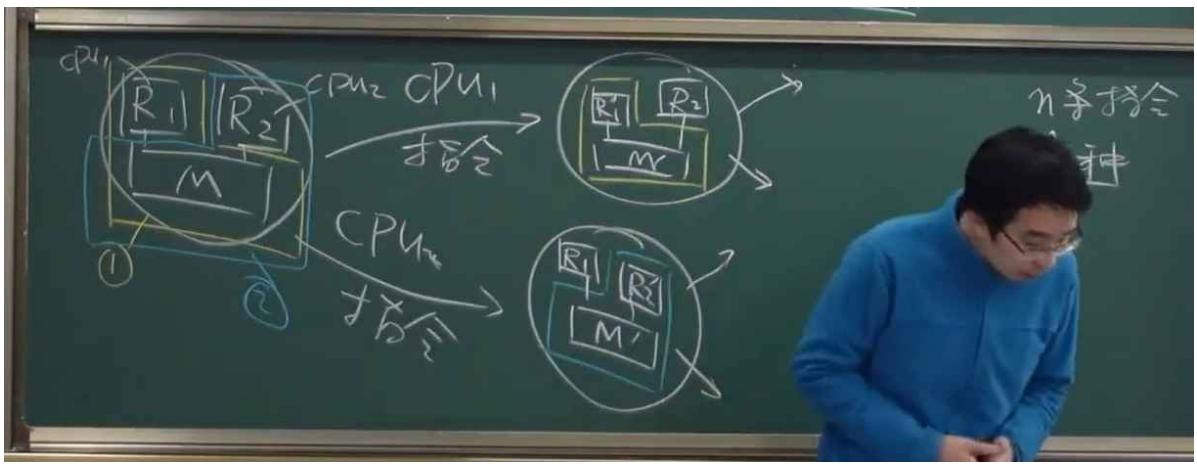


我们现在允许有多个cpu的存在，每个cpu都有一个自己的寄存器，然后大家共享一份内存：



这个时候我们再用前几节课的视角来看sys：

- 在前面的那个模型里面，在任何时候说机器是无情的执行指令的机器，计算机是无情执行指令的机器，那在多处理器机器上这点变成什么了呢？
- 变成了不确定的执行指令的机器，什么意思呢？我们可以把这样的一个多处理器计算机看成是两台单处理器计算机——R1+M和R2+M。
- 所以你可以想象，如果你把其他的处理器屏蔽掉，当那些处理器暂停，那么我们其实是得到了一个单处理器的计算机。在每一个时刻，我们可以选择哪个cpu来执行。



简易多处理器内核 (L1 的模型)

与多线程程序完全一致

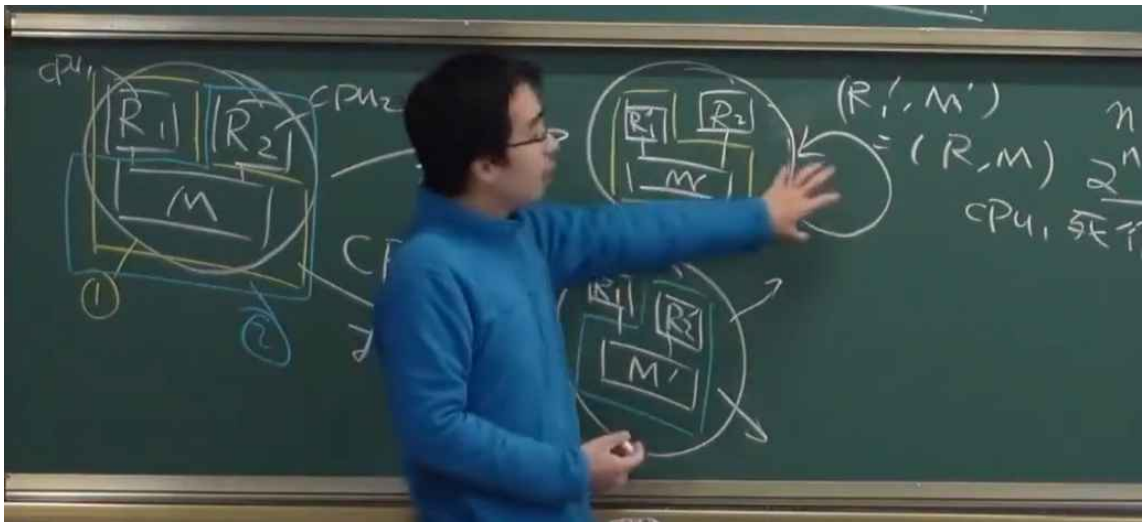
- 同步仅能通过原子指令如 xchg 实现

```
uint8_t shm[MEM_SIZE]; // Shared memory

void Tprocessor() {
  struct cpu_state s;
  while (1) {
    fetch_decode_exec(&s, shm);
  }
}

int main() {
  for (int i = 0; i < NPROC; i++) {
    create(Tprocessor);
  }
}
```

4 / 4

简化的模型就像上面这样子。这个简易的多处理器系统模型容易理解，但是也存在缺陷：它上面似乎没有办法允许任何“正经”的操作系统：如果任何处理器上的应用程序死循环，那么这个处理器就彻底”卡死“了。就相当于这个CPU从系统里面彻底消失了。但这不是我们对os的预期。

> 在刚开始学编程的时候，我们都会好奇，为什么我们写了一个死循环，我们不会把操作系统卡死？



而使真正我们线程得以”逃出“死循环的核心机制，就是操作系统中的**硬件中断**。

2. 中断机制

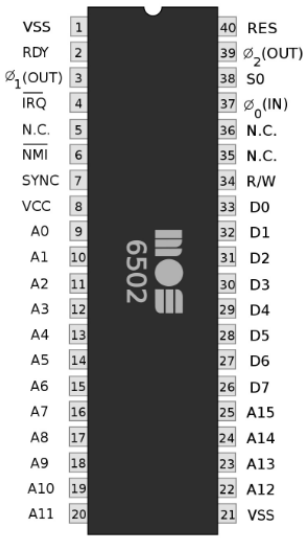
理解中断

硬件上的中断：一根线

- $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$ (边沿触发，低电平有效)
- “告诉处理器：停停，有事来了”
  - 剩下的行为交给处理器处理

实际的处理器并不是“无情地执行指令”

- 无情的执行指令
- 同时有情地响应外部的打断
  - 你在图书馆陷入了自习的死循环……
  - 清场的保安打断了你

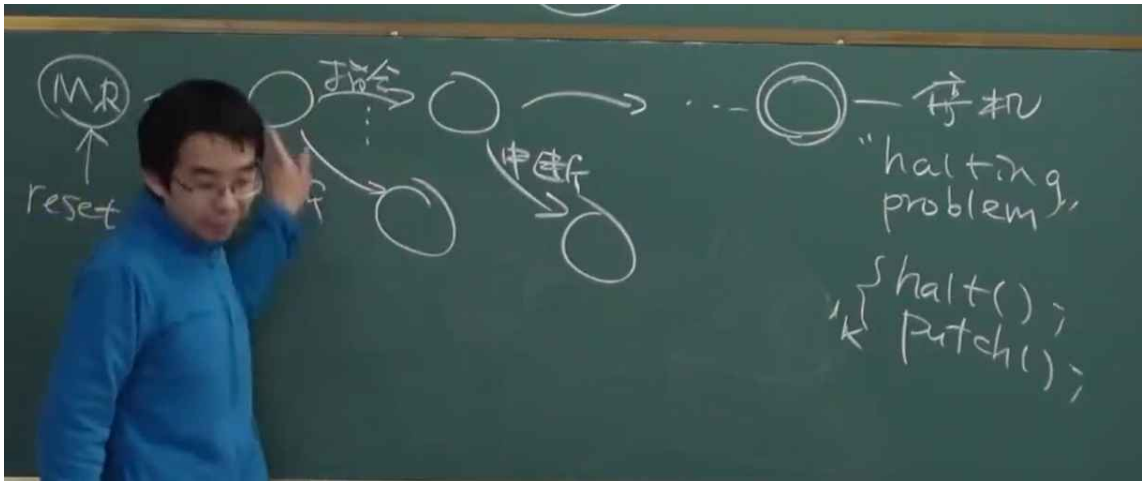


实际上中断也是很好理解的。你想如果你想要叫醒一个装睡的人，你怎么办？

死循环就是某一种程度的装睡，然后你要强行叫醒一个装睡的人，你就只能揍他。一定会给你一个response，所以所谓的中断实际上就是一个打断你当前所做的事情的一个机制。它的物理实现就是一根线。

他就是要打醒一个装睡的处理器，剩下的行为实际上是处理器定义的，就是处理器就可以收到这样的一个信号，相当于揍他一顿。

实际上我们的处理器并不是我们在计算机系统基础课上面学的这个简化的模型。实际上**处理器的真实的模型是，它在无情的执行指令的同时，它还会不断的响应外部的中断**。



虽然这样画并不非常严格，因为处理器有时候可以屏蔽某些中断。

> NMI 这个叫 non muscable interrupt，叫不可屏蔽中断，有些中断是可以屏蔽的，就比如说我觉得你太吵了，我就把你屏蔽了，对吧？我觉得你键盘太吵了，我把你屏蔽了，我不想，我不想收到来自键盘或者时钟这样的中断，这是做得到的。
>
> 但是有一些中断是不可屏蔽的，我举个例子，比如说 power loss，就这种非常紧急的事件，你最后可能有一个这个 rescue 的程序，它可以很短的程序，可以在你的电容还没有耗电量还没有耗尽的时候，可能执行一个几千个 cycle 的指令的时候，能做一些比较稳妥的安全操作，能使你的电脑不要受到太大的损害，这样的不可屏蔽的中断，但这都是这是非常少见的。

我们的 CPU 如果被打断，它应该干什么呢？如果你在图书馆陷入了自习的死循环，对吧？这个时候发生了一个中断——清场的保安打断了你。你应该进入另外一种模式去处理这个突发的事件。

处理器的中断行为

“响应”

- 首先检查处理器配置是否允许中断
  - 如果处理器关闭中断，则忽略
- x86 Family
  - 询问中断控制器获得中断号 n
  - 保存 CS, RIP, RFLAGS, SS, RSP 到堆栈
  - 跳转到 `IDT[n]` 指定的地址，并设置处理器状态 (例如关闭中断)
- RISC-V (M-Mode)
  - 检查 mie 是否屏蔽此次中断
  - 跳转 `PC = (mtvec & ~0xf)`
  - 更新 `mcause.Interrupt = 1`

不同的体系结构，它的中断处理程序能所做的事情就只有都是一个，就是一个跳转。因为中断相对于指令执行是一种比较罕见的事情。我们处理器显然对于罕见的事情应该有一个特殊的处理，当然这个特殊的处理又应该是软件来描述的。所以当中断到来的时候，我们的处理器就提供这样的一个机制，允许你去跑某些已经预先写好的代码。当然除了自动的跳转以外，它还会有一些处理器状态的设定。

3. 中断的意义

中断刚才我们也讲到，它就是在你死循环的时候，都可以把你可以强行在你的程序里面插入一个跳转，所以中断就是奠定操作系统霸主地位的一个机制。我们都知道所谓的操系统是把硬件资源给管起来，然后给应用程序提供服务，对吧？这句话当然没有讲错，那凭什么他能把硬件管起来？凭什么应用程序不能捣乱，对吧？就是因为中断机制在操作系统上看来和在应用程序上看来是不一样的，操作系统具有中断的完整的控制权，也就是说操系统想要打开中断它就打开，想要关闭中断它就关闭，但是应用程序是做不到的。

即使我们现在有一个死循环在处理器上跑，但是我们的系统每隔一段时间我们的时钟都会发出一个中断，所以无论你的代码是怎么样死循环的，总你执行那么几毫秒、几十毫秒以后总会收到一次中断，然后PC指向os代码，os kernal就开始执行了。

中断：奠定操作系统“霸主地位”的机制

操作系统内核 (代码)

- 想开就开，想关就关

应用程序

- 对不起，没有中断
  - 在 gdb 里可以看到 flags 寄存器 (`FL_IF`)
  - CLI — Clear Interrupt Flag
    - `#GP(0)` If CPL is greater than IOPL and less than 3
    - 试一试 `asm volatile ("cli");`
- 死循环也可以被打断

如果应用程序试图把中断关掉呢？当我编译运行它的时候，它就直接给了我一个 segmentation fault。总之就是在你的应用程序在运行的时候，我们的计算机提供了这样的一个机制，当你干出这么一个骚操作，它会抛出一个十三号的异常。就是当我执行那个 CRI 指令的时候，我的处理器就会产生一个内部的中断。然后我的操作系统收到这个中断，就知道该程序执行的非法操作。以前早期的时候，Windows 程序如果出错的话，他都会说该程序执行的非法操作即将被关闭，所以这个程序就会被中断。

AbstractMachine 中断 API

隔离出一块“处理事件”的代码执行时空

- 执行完后，可以返回到被中断的代码继续执行

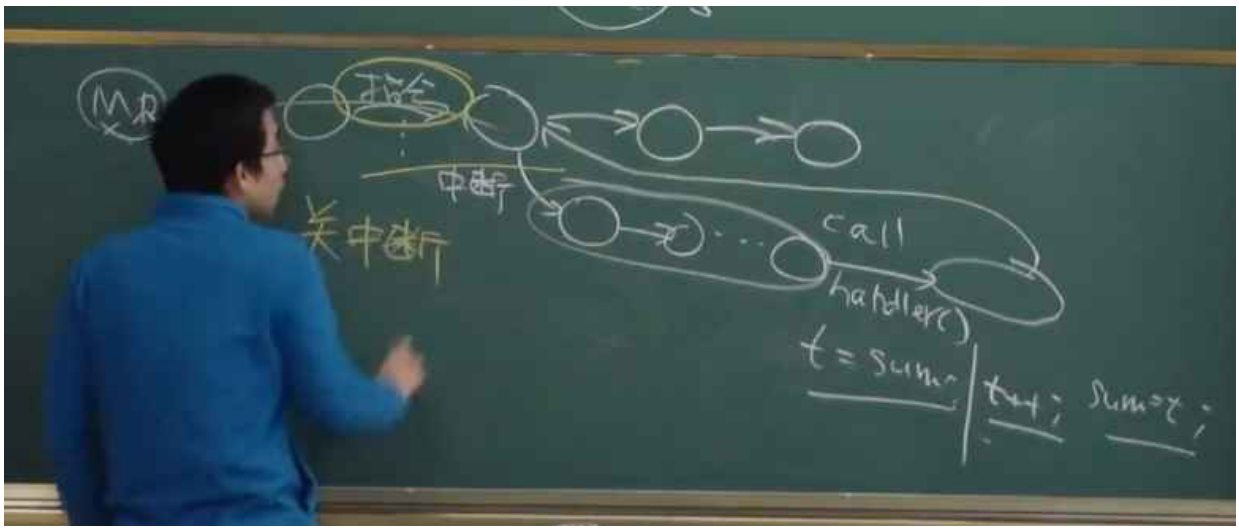
```
bool cte_init(Context *(*handler)(Event ev, Context *ctx));
bool ienabled(void);
void iset(bool enable);
```

中断引入了并发

- 最早操作系统的并发性就是来自于中断 (而不是多处理器)
- 关闭中断就能实现互斥
  - 系统中只有一个处理器，永远不会被打断
  - 关中断实现了“stop the world”

我们lab中的api：

- 我们可以在处理器初始化的时候，整个系统初始化的时候执行一个cte_init(context)。这个函数指针你只要传一个函数，然后这个函数它接受两个参数，一个参数是一个event，还有一个是这个中断时候的当前的context。
- 同时我们可以询问当前中断是打开还是关闭的，对吧？它返回出或者false，这样代表当前中断处理器是响应中断还是不响应中断。
- enable 或者disable中断机制。



如果关了中断，你的处理器就永远不会被打断。对于单处理器系统来说，关闭中断就可以实现互斥——永远不会有并发了。

真正的计算机系统模型

状态

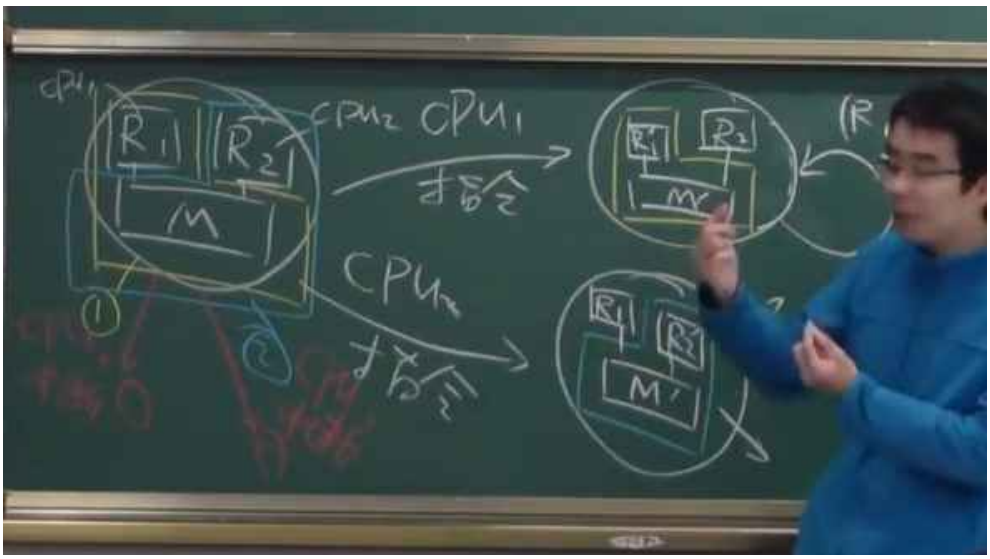
- 共享内存和每个处理器的内部状态 $(M, R_1, R_2, \ldots, R_n)$

状态迁移

1. 处理器 t 执行一步 $(M, R_t) \to (M', R_t')$
   - 新状态为 $(M', R_1, R_2, \ldots, R_t', \ldots, R_n)$
2. 处理器 t 响应中断

假设 race-freedom

- 不会发生 relaxed memory behavior
- 模型能触发的行为就是真实系统能触发的所有行为



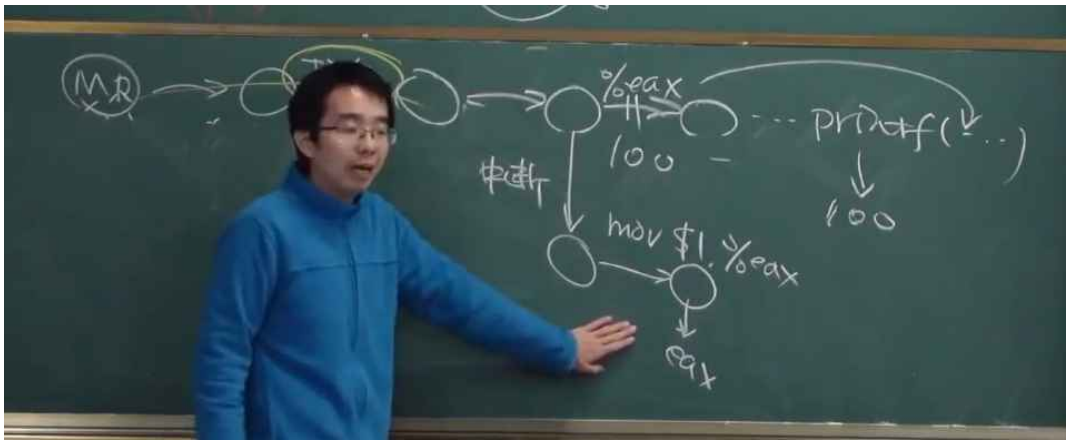
2. 50行实现嵌入式操作系统

中断处理程序的秘密

参数和返回值“Context”

```
Context *handler(Event ev, Context *ctx) {
  ...
}
```

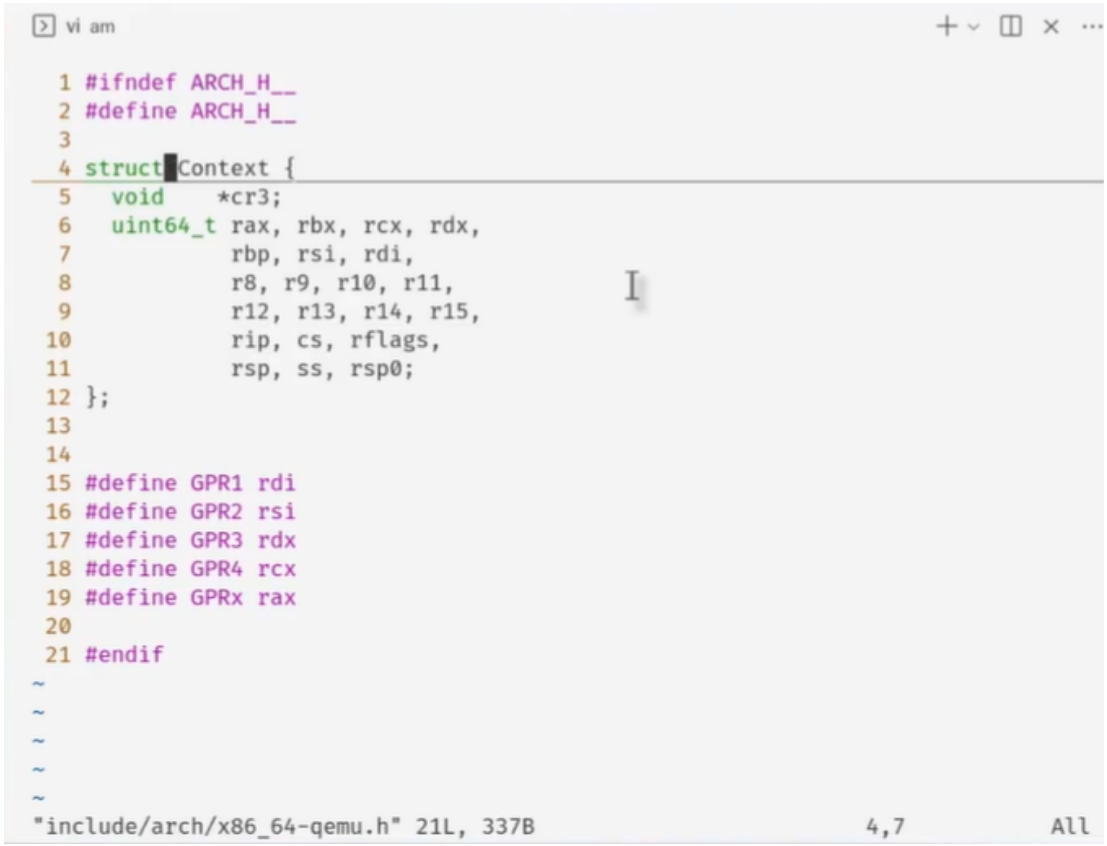
- 中断发生后，不能执行“任何代码”
  - `movl $1, %rax` 先前状态机就被永久“破坏”了
  - 除非把中断瞬间的处理器状态保存下来
  - 中断返回时需要把寄存器恢复到处理器上
- 看看 Context 里有什么吧



上下文切换

状态机在执行……

- 发生了中断
  - 操作系统代码开始执行
  - 状态机被“封存”
  - 我们可以用代码“切换”到另一个状态机
- 中断返回另一个状态机的状态
  - “封存”的状态还在
  - 下次可以被恢复到处理器上执行



Take away messages

本节课遭遇了画风突变：在做了 “状态机” 和 “并发” 足够的铺垫后，我们终于回到了机器指令 (状态机) 和操作系统内核了。本次课回答了为什么 `while (1)` 不会把操作系统 “卡死”：

- 在操作系统代码切换到应用程序执行时，操作系统内核会打开中断
- “不可信任” 的应用程序定期会收到时钟中断被打断，且应用程序无权配置中断
- 操作系统接管中断后，可以切换到另一个程序执行