# Context-switch & Threads

# Goal of Today's Class

- Understand the concepts of context, context-switch and threads
- Understand the related functions in assignment P1
    - `thread_init, thread_create, thread_yield, thread_exit`
    - `ctx_entry, ctx_start, ctx_switch`

Review: the minimal requirement of program execution is code & stack segments in memory address space.

# Assume 2 programs in memory

program#1 stack

program#1 code

program#2 stack

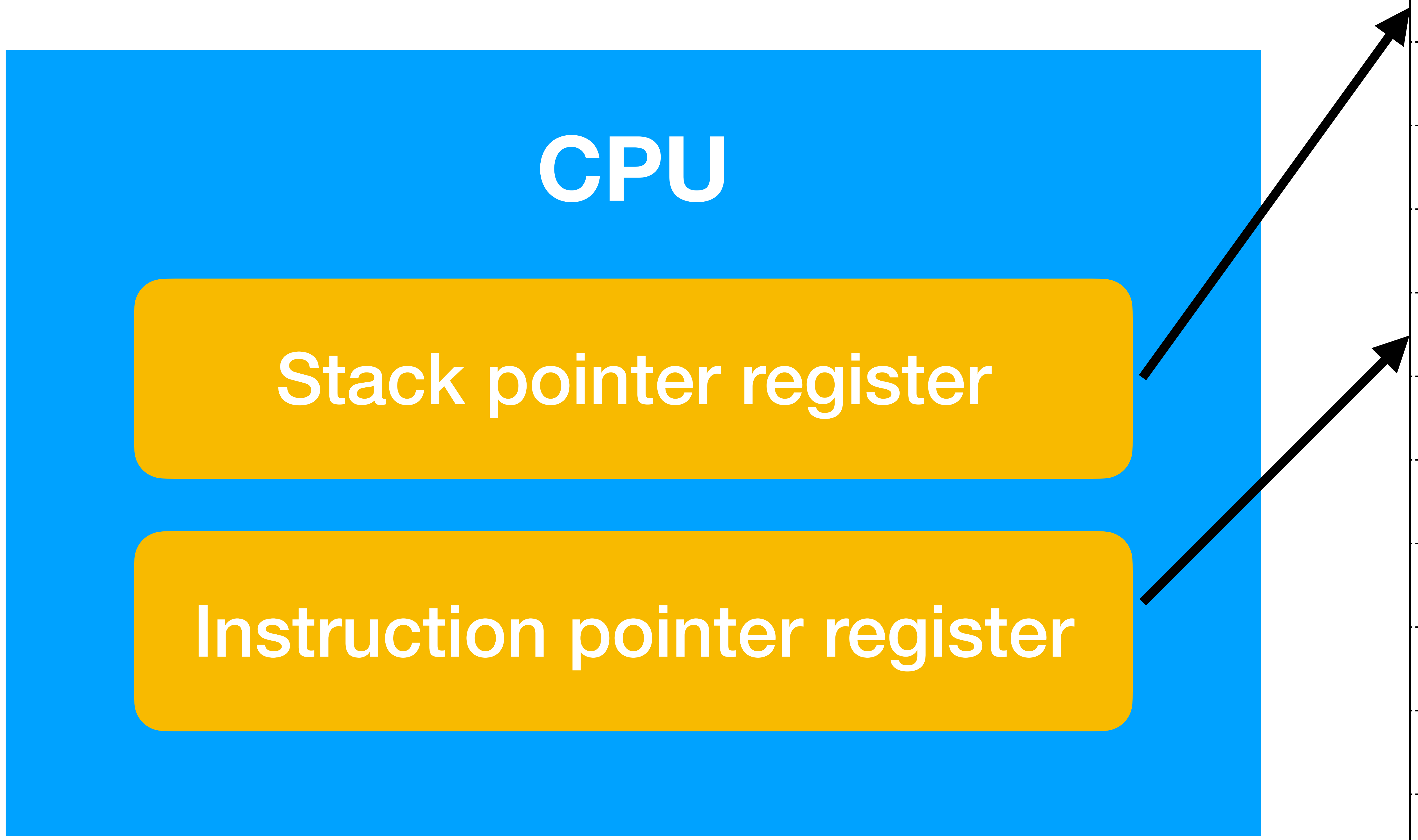
program#2 code

| | |
|---|---|
| stack end | … |
| … | … |
| stack start | … |
| … | … |
| code end | … |
| … | … |
| code start | … |
| … | … |
| stack end | … |
| … | … |
| stack start | … |
| … | … |
| code end | … |
| … | … |
| code start | … |

OS puts the code & stack of both programs in the memory so that they can take turns to execute.

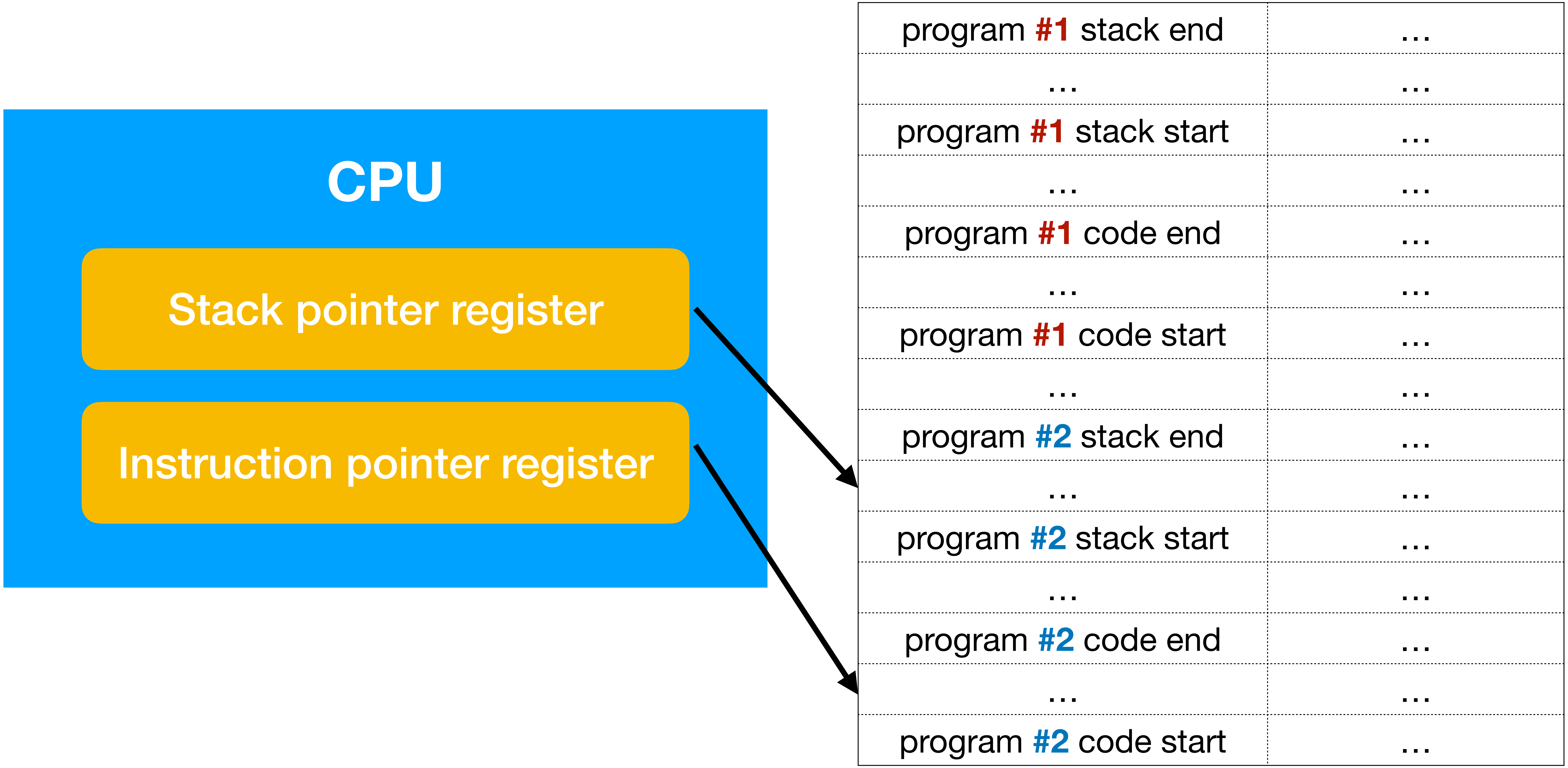
Review: context defines which program the CPU is executing; context = memory address space + stack pointer + instruction pointer

# CPU in the context of program #1



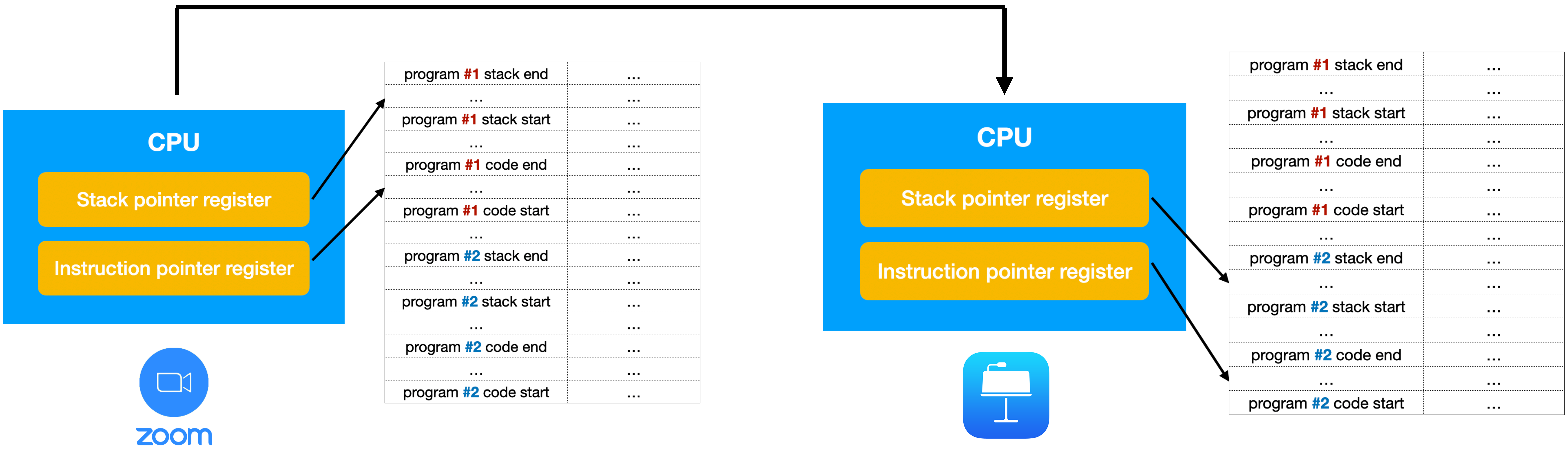

| | |
|---|---|
| program #1 stack end | … |
| … | … |
| program #1 stack start | … |
| … | … |
| program #1 code end | … |
| … | … |
| program #1 code start | … |
| … | … |
| program #2 stack end | … |
| … | … |
| program #2 stack start | … |
| … | … |
| program #2 code end | … |
| … | … |
| program #2 code start | … |

# CPU in the context of program #2

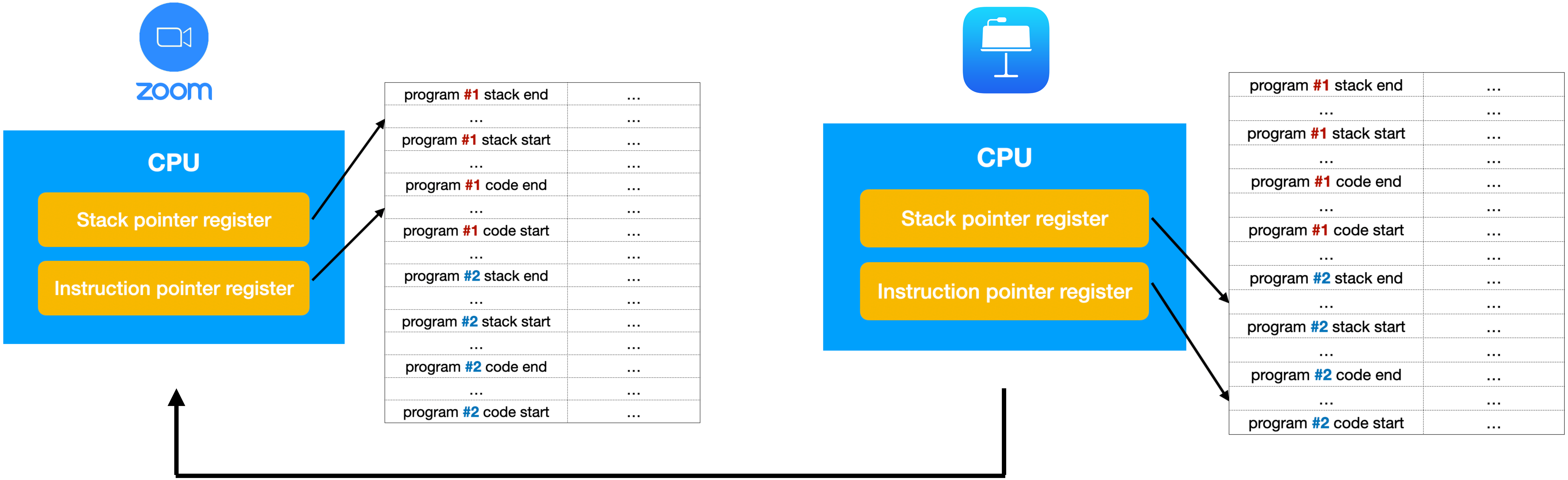


# Context-switch

**CPU switches to the context of program #2**



# Context-switch




**CPU switches to the context of program #1**

**Question**: when does context-switch happen?

# When does context-switch happen?

- Program terminates.
- Program calls `yield` system call. (next slide)
- CPU receives a timer interrupt. (later in assignment P2)
- CPU receives an I/O interrupt. (later in assignment P5)

# yield is a noble behavior


A car can occupy the road, but it decides to stop and let others to use the road first.

```
int noble_a() {
    ......
    yield();
    ......
}
```

A program can occupy the CPU, but it decides to stop and let others to use the CPU first.

# Two noble functions

```
int noble_a() {
    printf(“Noble A does some work”);
    ……
    yield();
    printf(“Noble A works some more”);
    ……
}
```

```
int noble_b() {
    printf(“Noble B does some work”);
    ……
    yield();
    printf(“Noble B works some more”);
    ……
}
```

# One possible schedule

```
int noble_a() {
 1 printf(“Noble A does some work”);
   ……
 2 yield();
   printf(“Noble A works some more”);
 5 ……
}
```

```
int noble_b() {
 3 printf(“Noble B does some work”);
   ……
 4 yield();
   printf(“Noble B works some more”);
 6 ……
}
```

```
Output:
Noble A does some work
Noble B does some work
Noble A works some more
Noble B works some more
```

# Another possible schedule

```
int noble_a() {
3  printf("Noble A does some work");
   ......
4  yield();
   printf("Noble A works some more");
6  ......
}
```

```
int noble_b() {
1  printf("Noble B does some work");
   ......
2  yield();
   printf("Noble B works some more");
5  ......
}
```

```
Output:
Noble B does some work
Noble A does some work
Noble B works some more
Noble A works some more
```

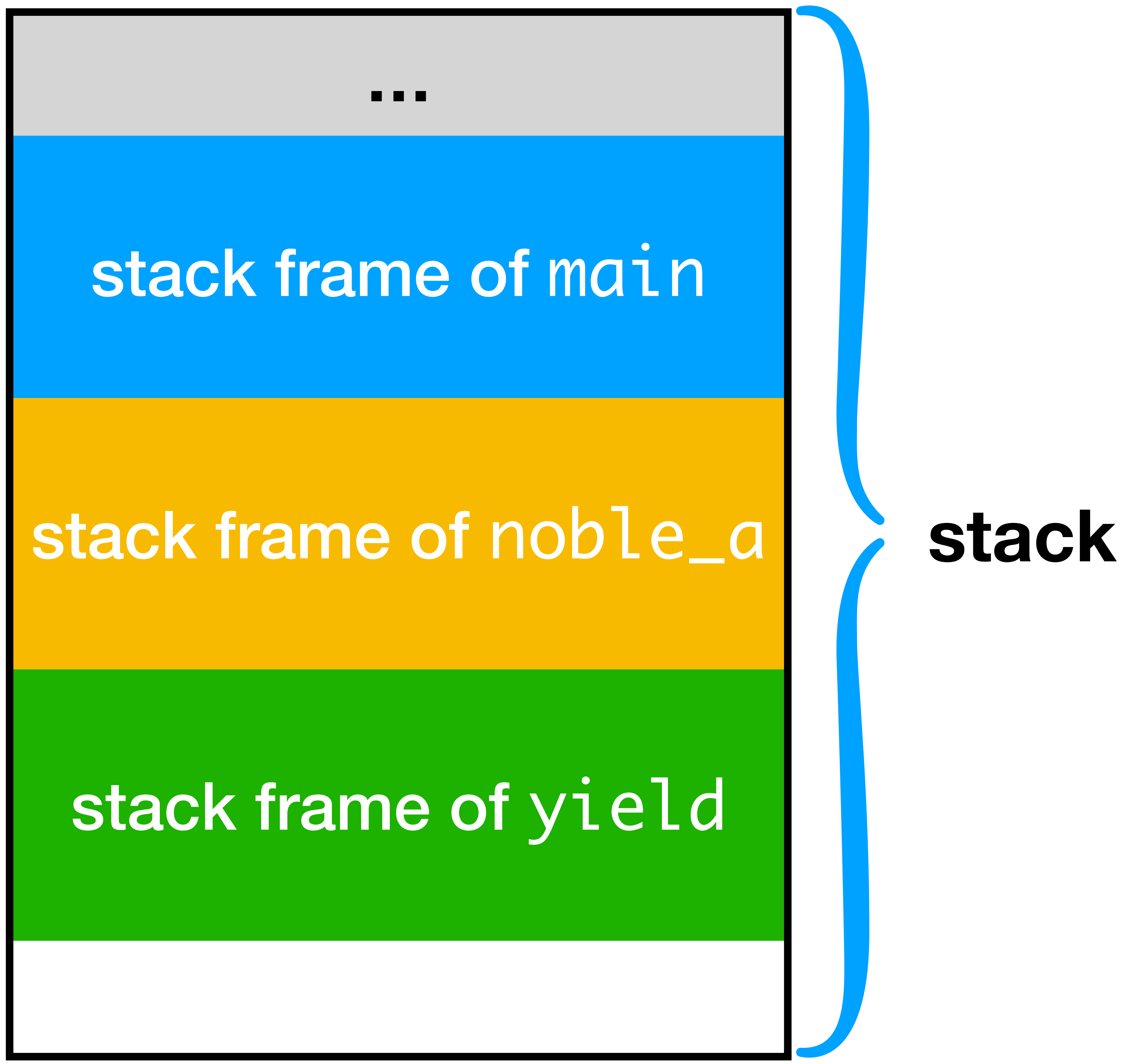
# Question: how do we run two functions at the same time?

**Let’s review some knowledge of stack.**

# Review of stack

```
int noble_a() {
    printf("Noble A does some work");
    ……
    yield();
    printf("Noble A works some more");
    ……
}

int main() {
    noble_a();
    return 0;
}
```



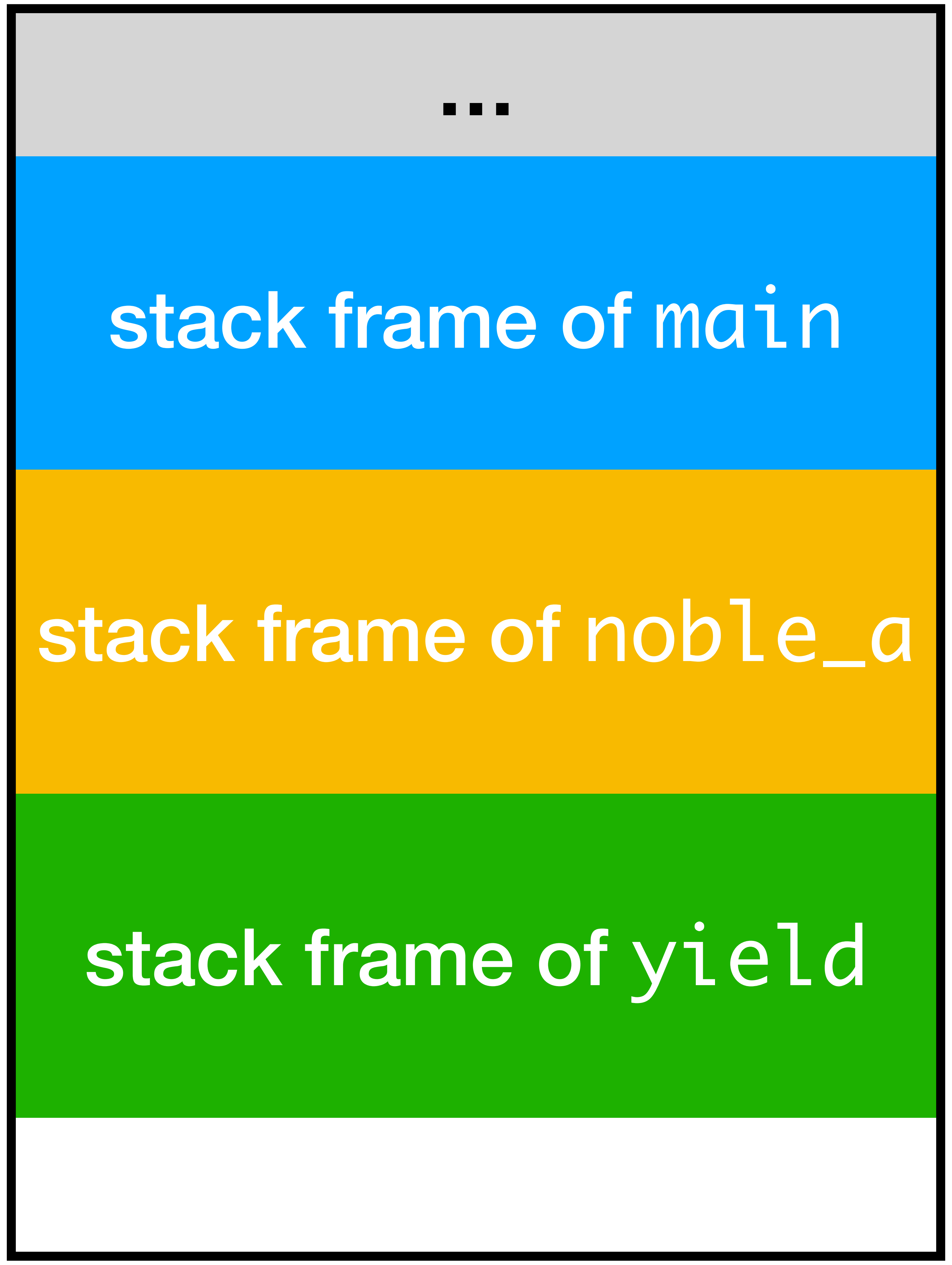

main() calls noble_a() calls yield()

# Review of stack

**Only continue when** noble_a **returns**

**Only continue when** yield **returns**

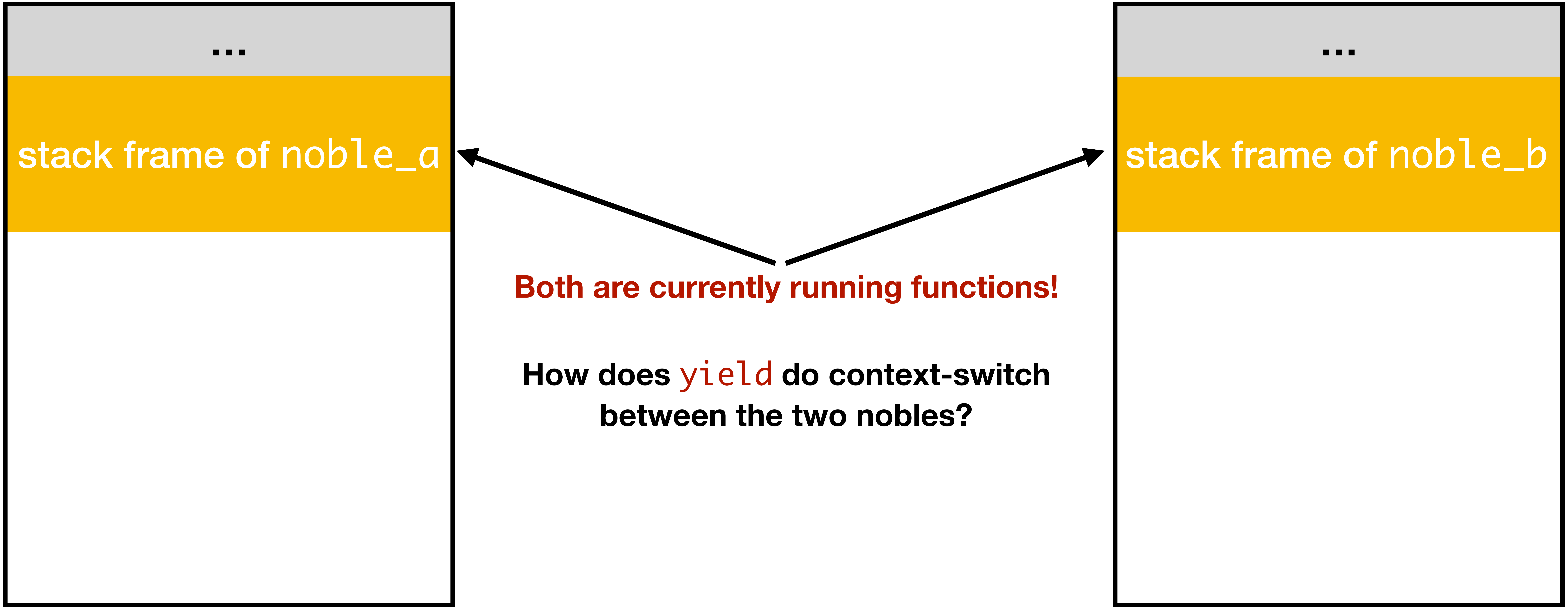
**Currently running function**




For a single stack, there is **only one** currently running function, so that noble_a and noble_b cannot run in the same stack at the same time.

`main() calls noble_a() calls yield()`

# Two stacks for two nobles



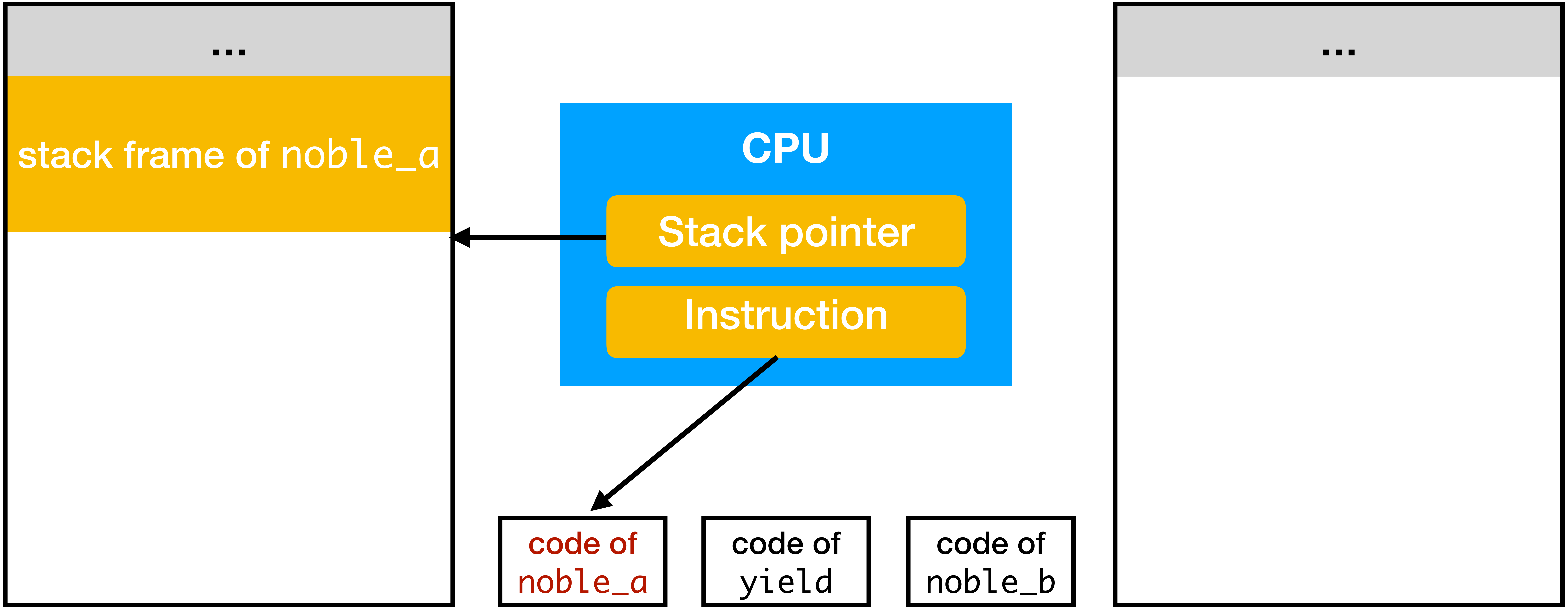
# Noble A does some work

```
int noble_a() {
1   printf(“Noble A does some work”);
    ……
    yield();
    printf(“Noble A works some more”);
    ……
}
```

```
int noble_b() {
    printf(“Noble B does some work”);
    ……
    yield();
    printf(“Noble B works some more”);
    ……
}
```

Output:
Noble A does some work

# CPU in context of noble_a



# Noble A yields

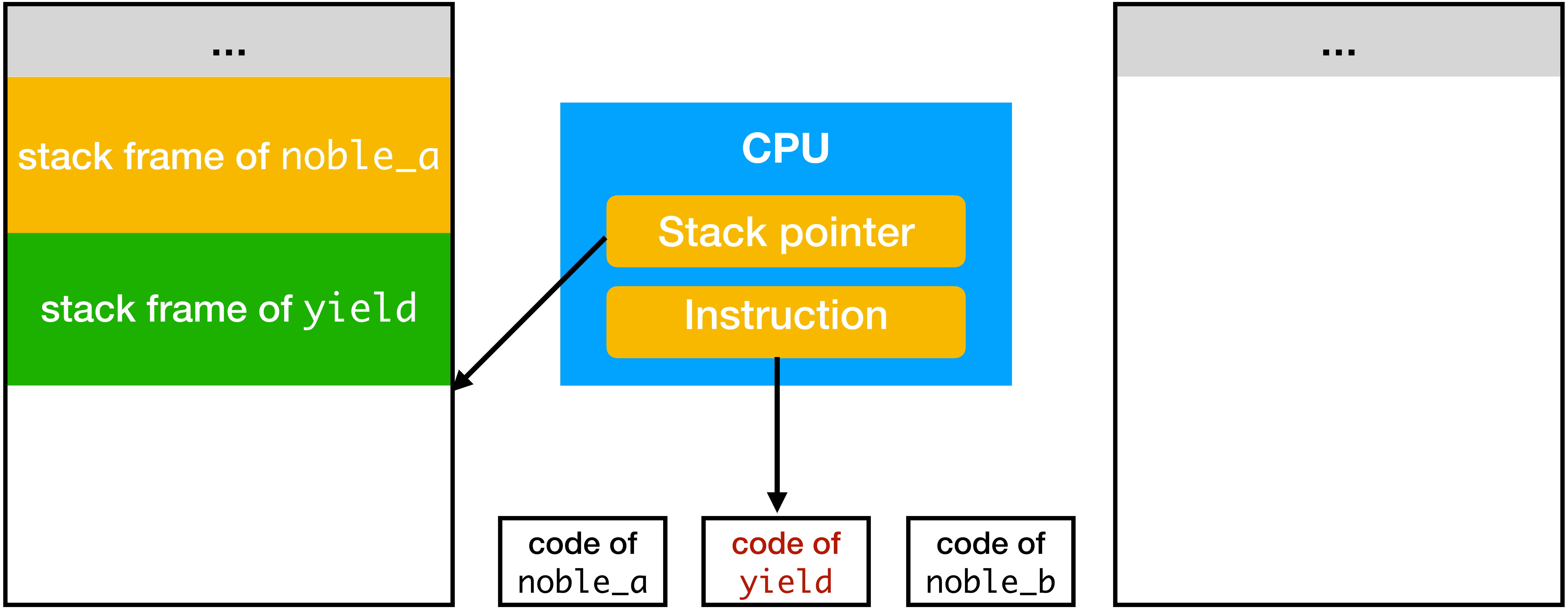
```
int noble_a() {
1   printf("Noble A does some work");
    ......
2   yield();
    printf("Noble A works some more");
    ......
}
```

```
int noble_b() {
    printf("Noble B does some work");
    ......
    yield();
    printf("Noble B works some more");
    ......
}
```

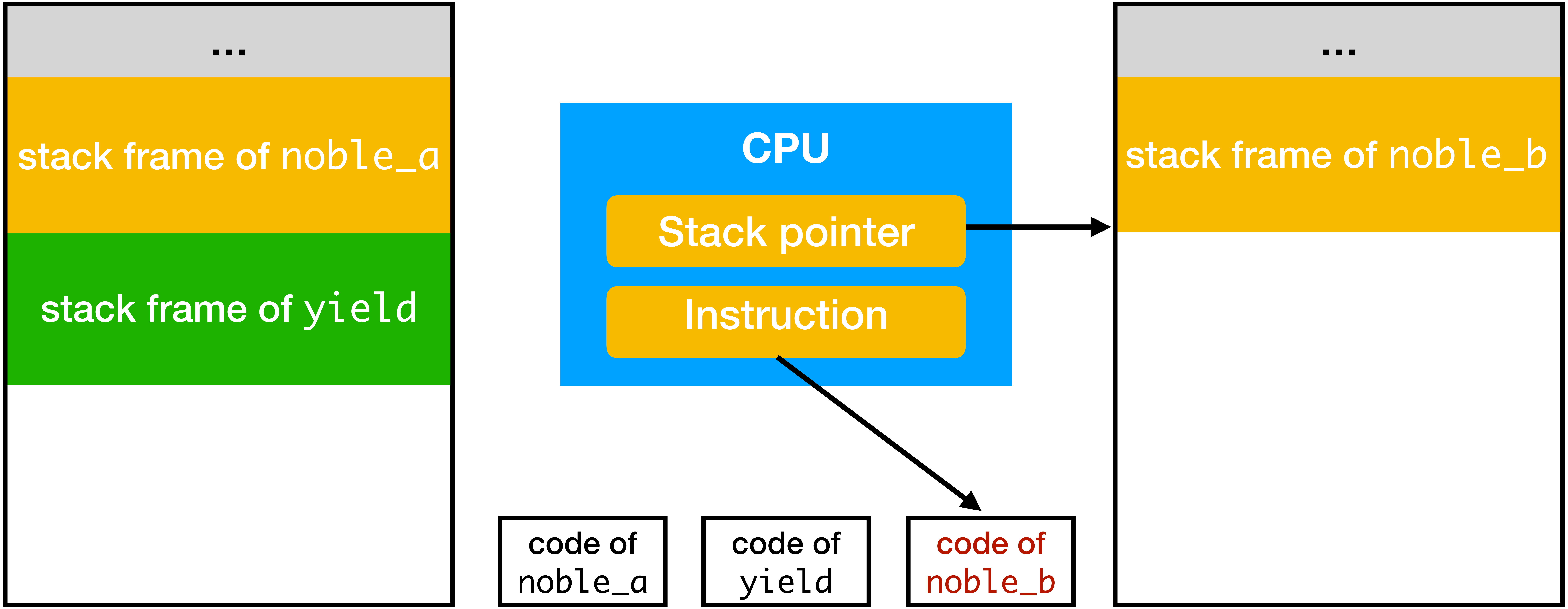
Output:

Noble A does some work

# noble_a calls yield



# yield switches context to noble_b



# Noble B does some work

```
int noble_a() {
1  printf("Noble A does some work");
   ……
2  yield();
   printf("Noble A works some more");
   ……
}
```

```
int noble_b() {
3  printf("Noble B does some work");
   ……
   yield();
   printf("Noble B works some more");
   ……
}
```

```
Output:
Noble A does some work
Noble B does some work
```

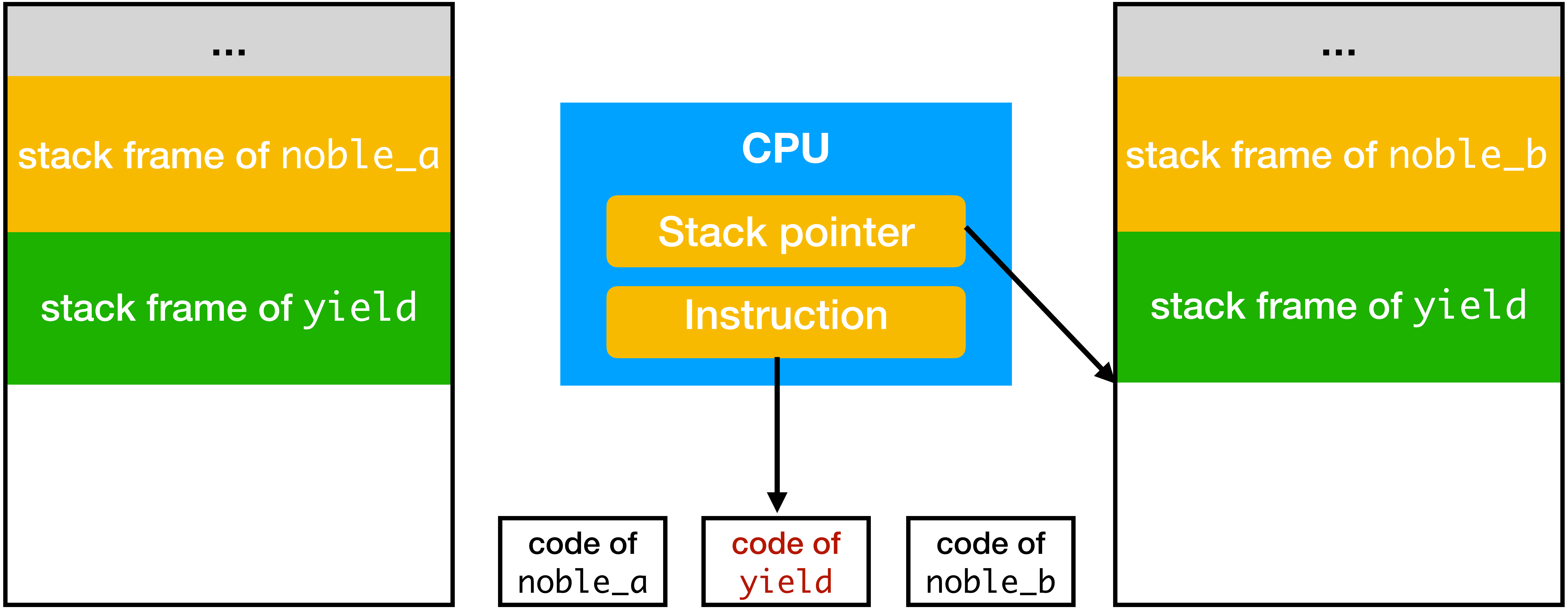
# Noble B yields

```
int noble_a() {
 1 printf("Noble A does some work");
   ......
 2 yield();
   printf("Noble A works some more");
   ......
}
```

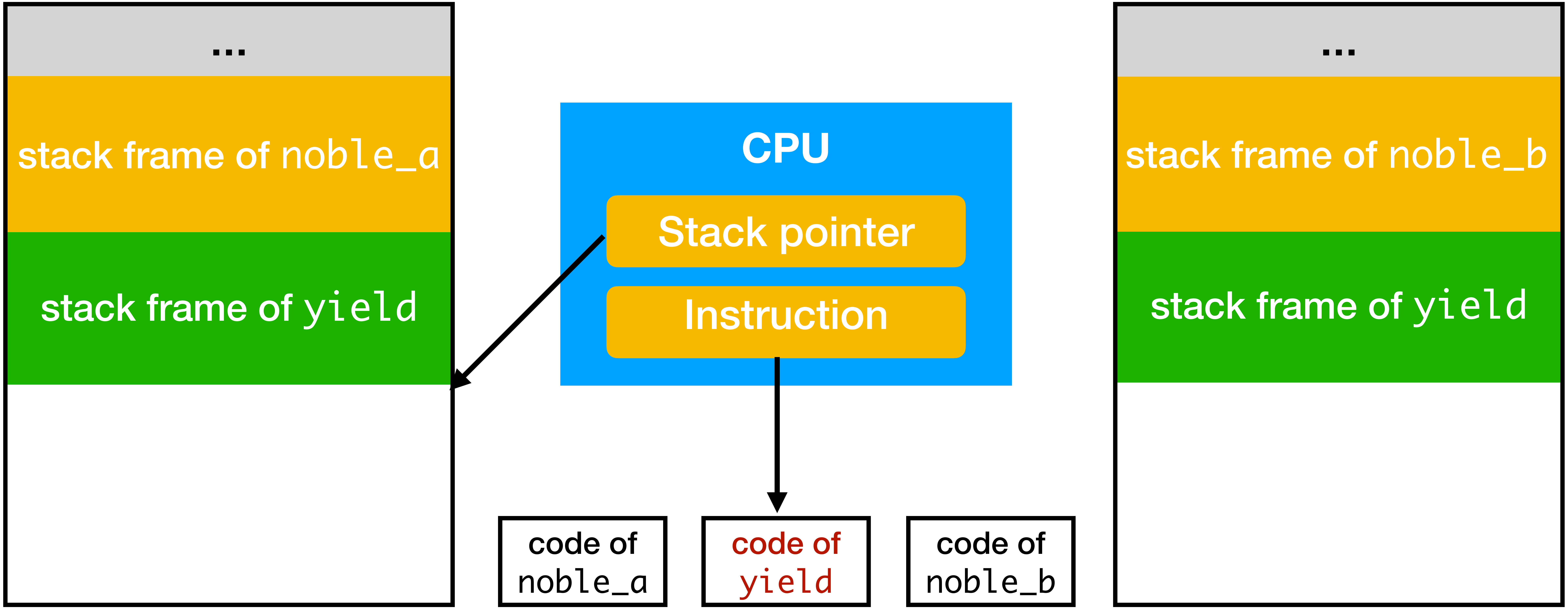
```
int noble_b() {
 3 printf("Noble B does some work");
   ......
 4 yield();
   printf("Noble B works some more");
   ......
}
```

```
Output:
Noble A does some work
Noble B does some work
```

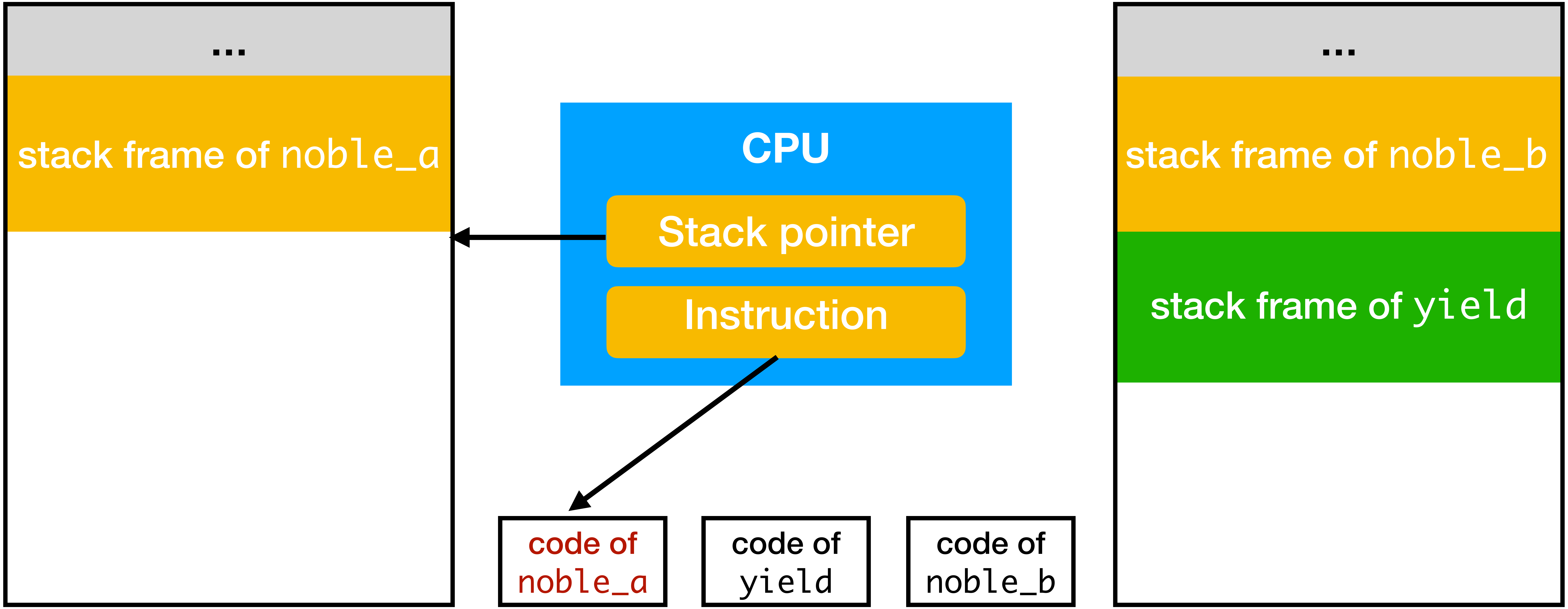
# noble_b calls yield



# yield switches context to noble_a



# yield returns to noble_a



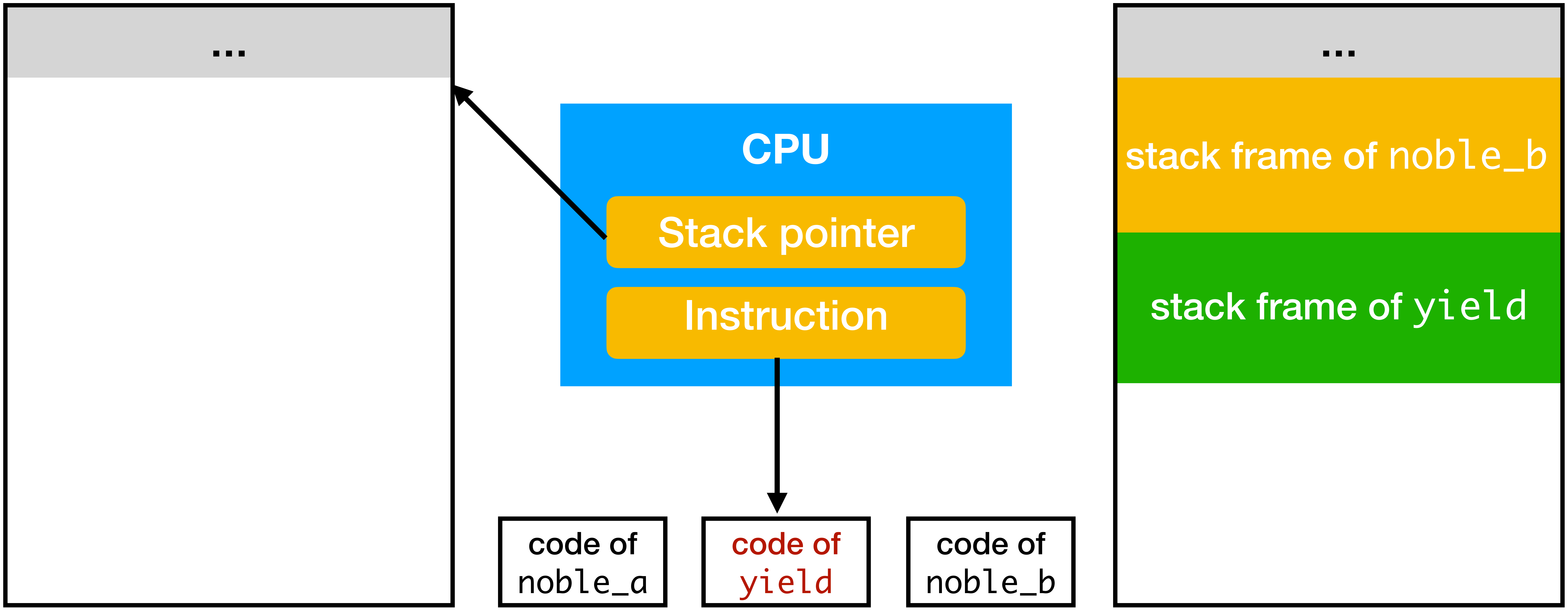
# Noble A works some more

```
int noble_a() {
1  printf("Noble A does some work");
   ......
2  yield();
   printf("Noble A works some more");
5  ......
}
```

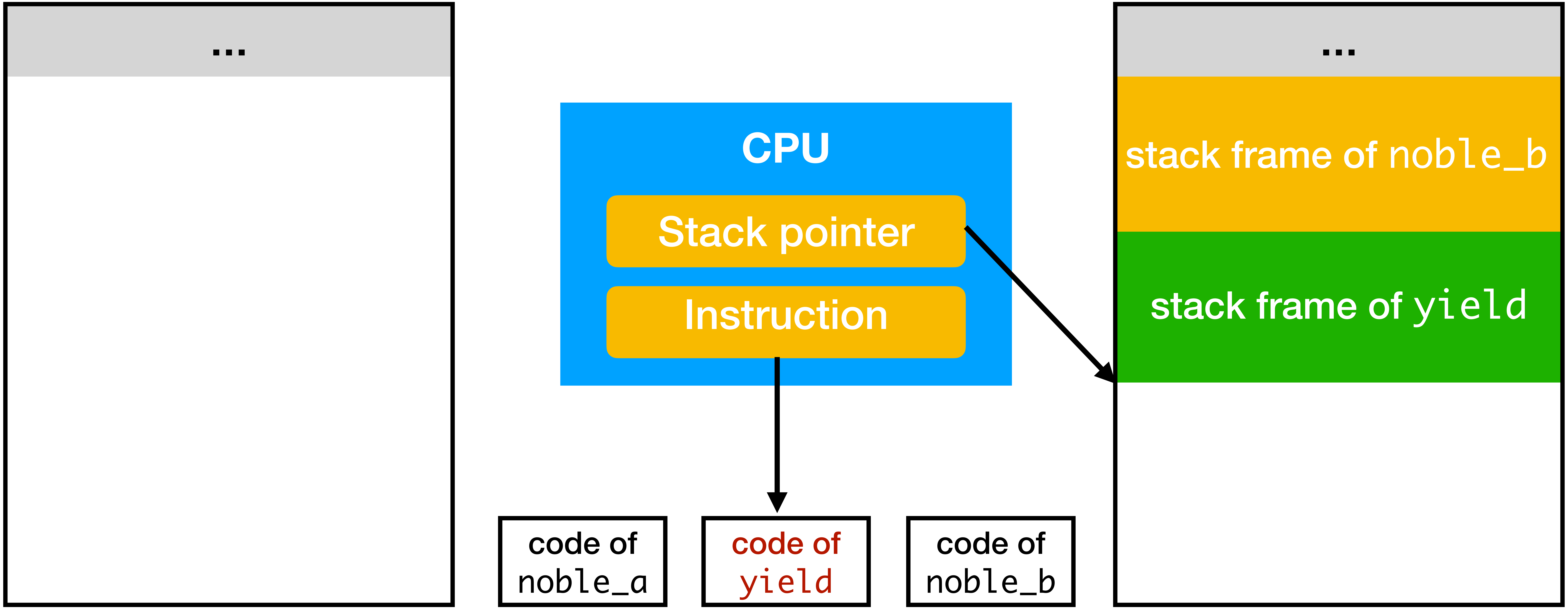
```
int noble_b() {
3  printf("Noble B does some work");
   ......
4  yield();
   printf("Noble B works some more");
   ......
}
```

```
Output:
Noble A does some work
Noble B does some work
Noble A works some more
```

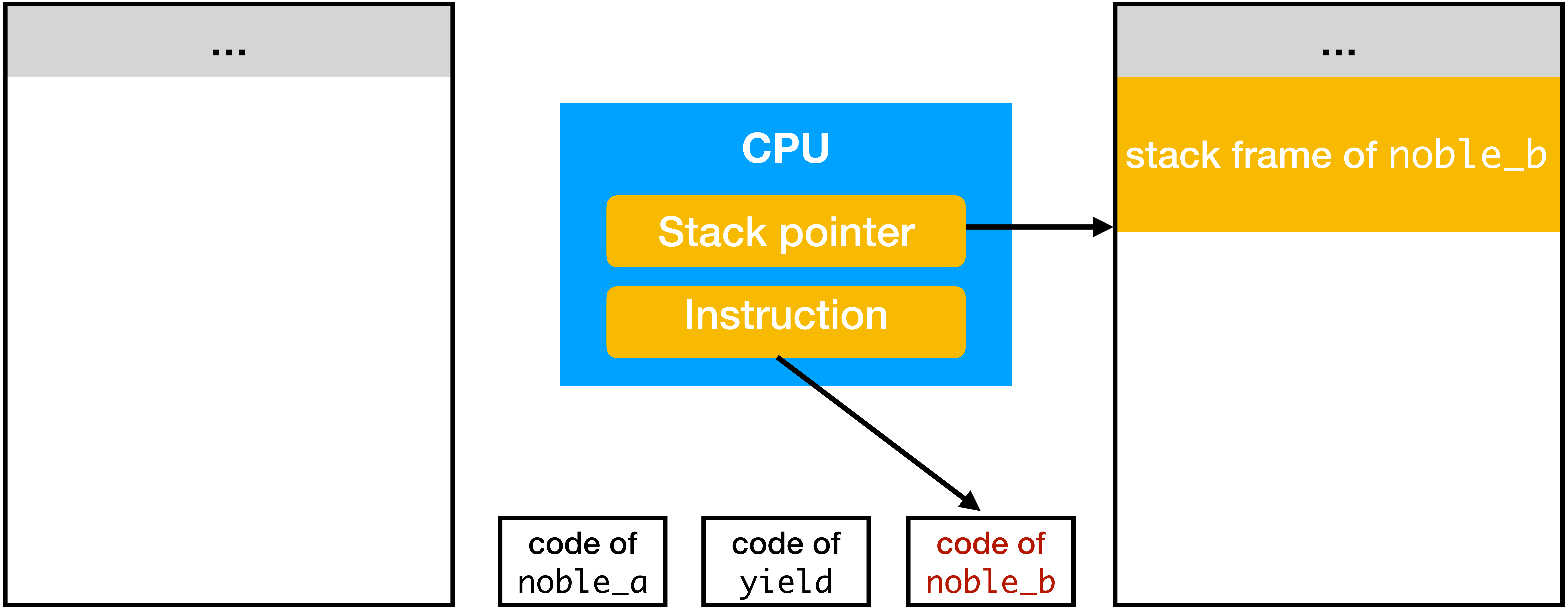
# noble_a terminates (implicit yield)



# yield switches context to noble_b



# yield returns to noble_b



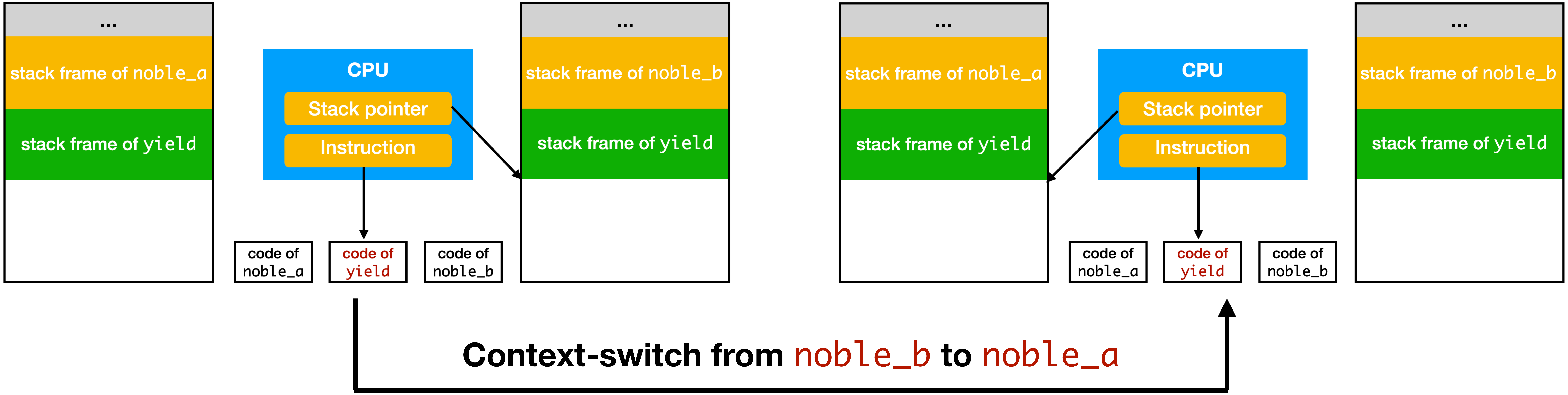
# Noble B works some more

```
int noble_a() {
1 printf("Noble A does some work");
  ......
2 yield();
  printf("Noble A works some more");
5 ......
}
```

```
int noble_b() {
3 printf("Noble B does some work");
  ......
4 yield();
  printf("Noble B works some more");
6 ......
}
```

```
Output:
Noble A does some work
Noble B does some work
Noble A works some more
Noble B works some more
```

# How does yield do context-switch?




- The answer is simple: change the stack pointer!
  - when switching from noble_a to noble_b, the `yield` function needs to record the stack pointer of noble_a
  - when switching back to noble_a, the `yield` function restores the stack pointer of noble_a

# Recall: when does context-switch happen?

✅ Program terminates.

✅ Program calls `yield` system call.

- CPU receives a timer interrupt. (later in assignment P2)
- CPU receives an I/O interrupt. (later in assignment P5)

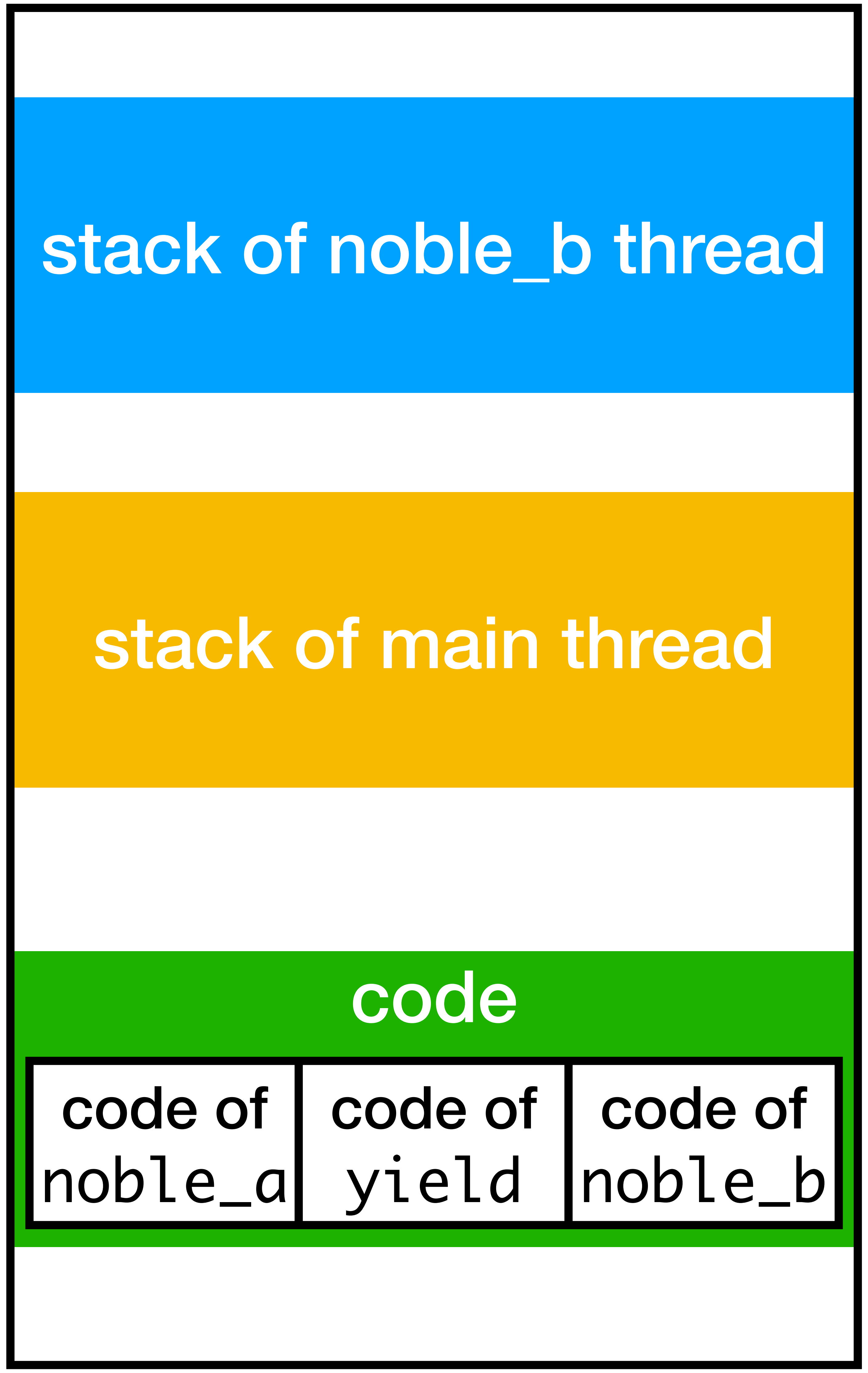
**Lesson**: A **thread** owns a stack running a given function.

# A thread owns a stack running a given function

```
int main() {
    thread_create(noble_b);
    noble_a();
    return 0;
}

int noble_a() {
    ……
}

int noble_b() {
    ……
}
```

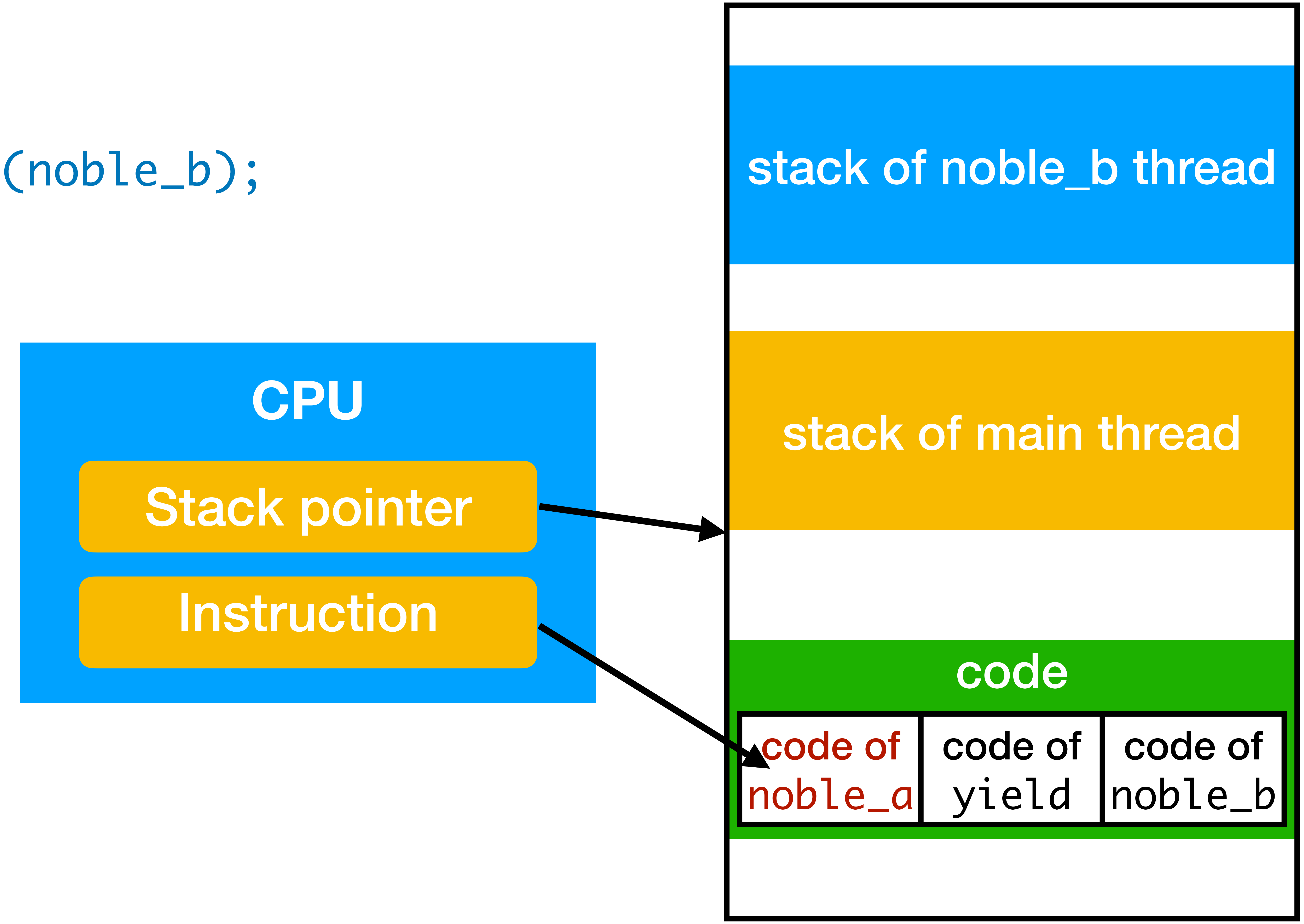


# Context-switch between threads

```
int main() {
    thread_create(noble_b);
    noble_a();
    return 0;
}

int noble_a() {
    ......
}

int noble_b() {
    ......
}
```

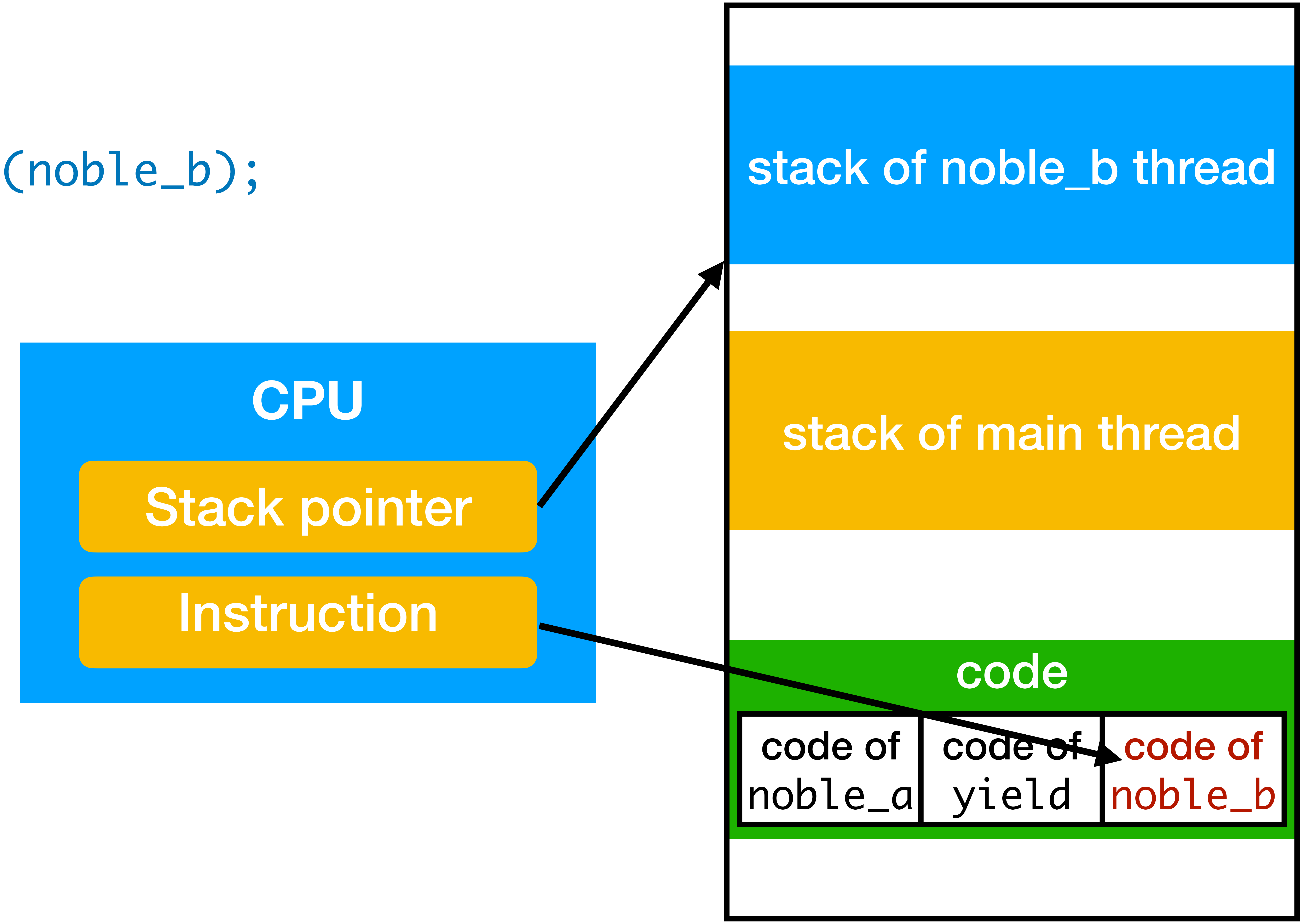


# Context-switch between threads

```
int main() {
    thread_create(noble_b);
    noble_a();
    return 0;
}

int noble_a() {
    ……
}

int noble_b() {
    ……
}
```



# Goal of Today's Class

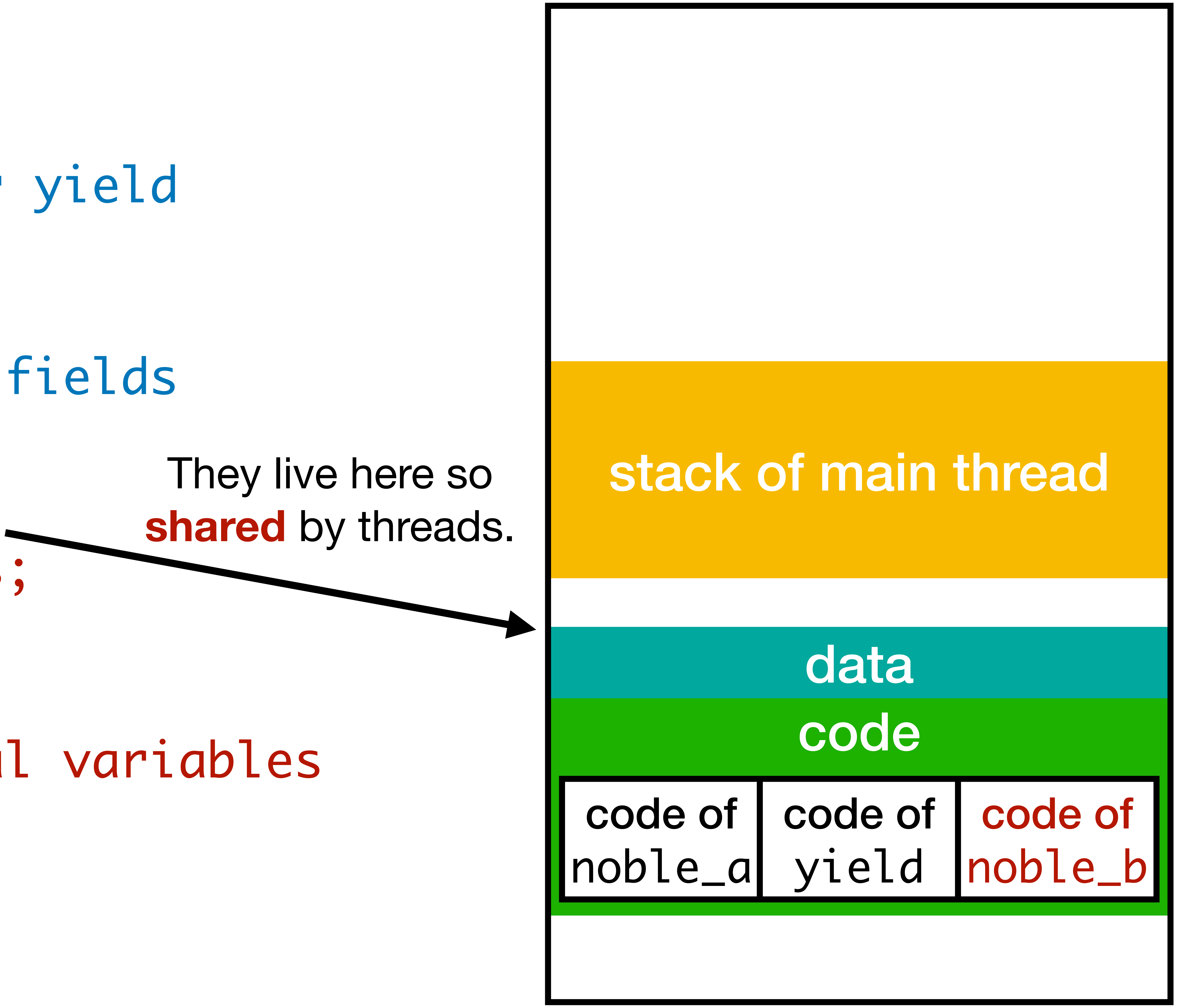
- ✅ Understand the concepts of context, context-switch and threads
- Understand the related functions in assignment P1
  - `thread_init`, `thread_create`, `thread_yield`, `thread_exit`
  - `ctx_entry`, `ctx_start`, `ctx_switch`

# thread_init

```
struct thread {
    // stored stack pointer for yield
    // *func to call
    // *arg to the function
    // …… feel free to add new fields
};
struct thread current_thread;
struct queue_t runnable_threads;

int main() {
    // initialize the two global variables
    thread_init();
    return 0;
}
```

They live here so **shared** by threads.

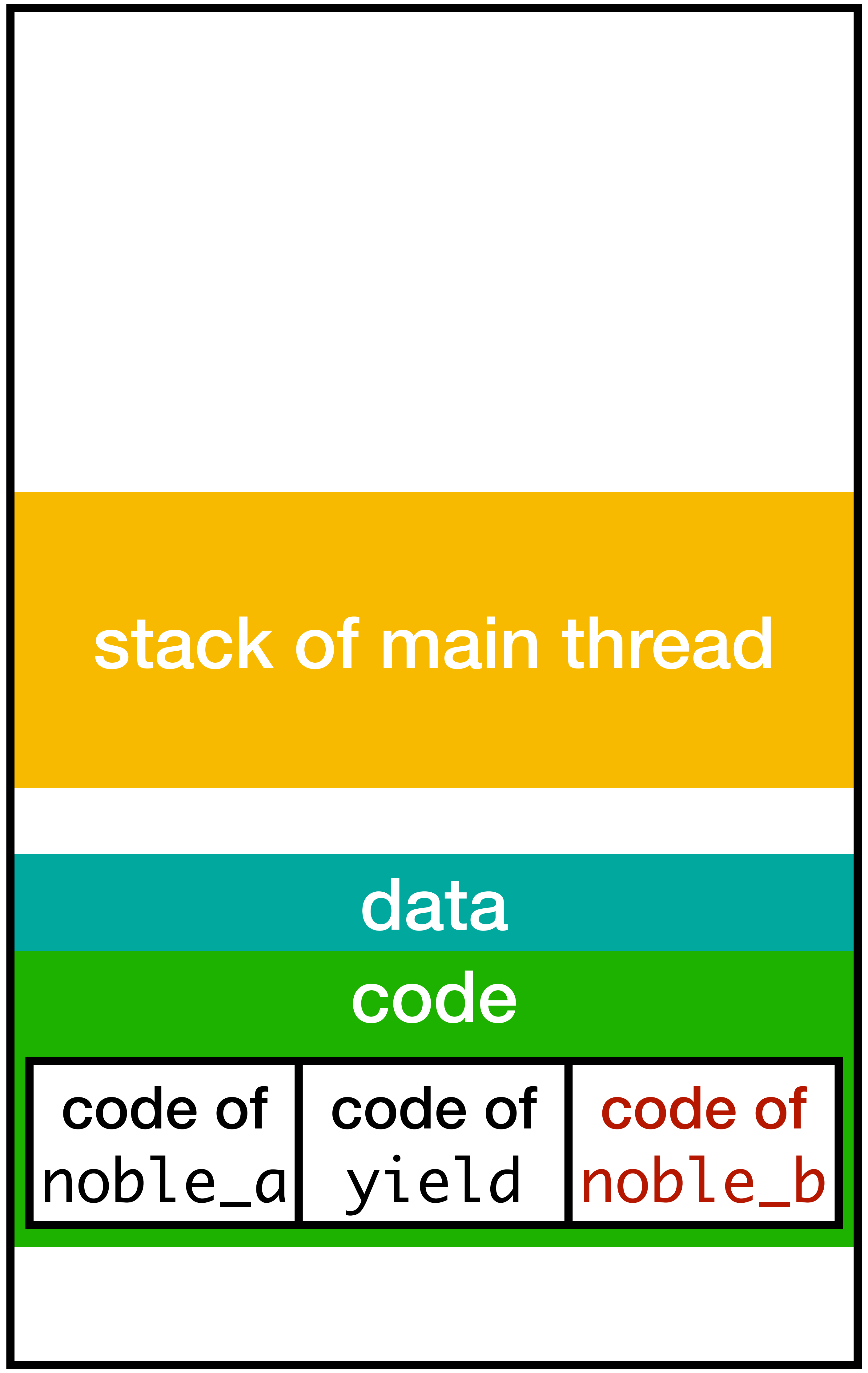


# thread_create

```
struct thread current_thread;
struct queue_t runnable_threads;

void noble_b(void* arg) {
    ……
}

int main() {
    // initialize the two global variables
    thread_init();
    // create a thread by modifying
    // the global variables
    thread_create(noble_b, NULL, 16 * 1024);
    return 0;
}
```

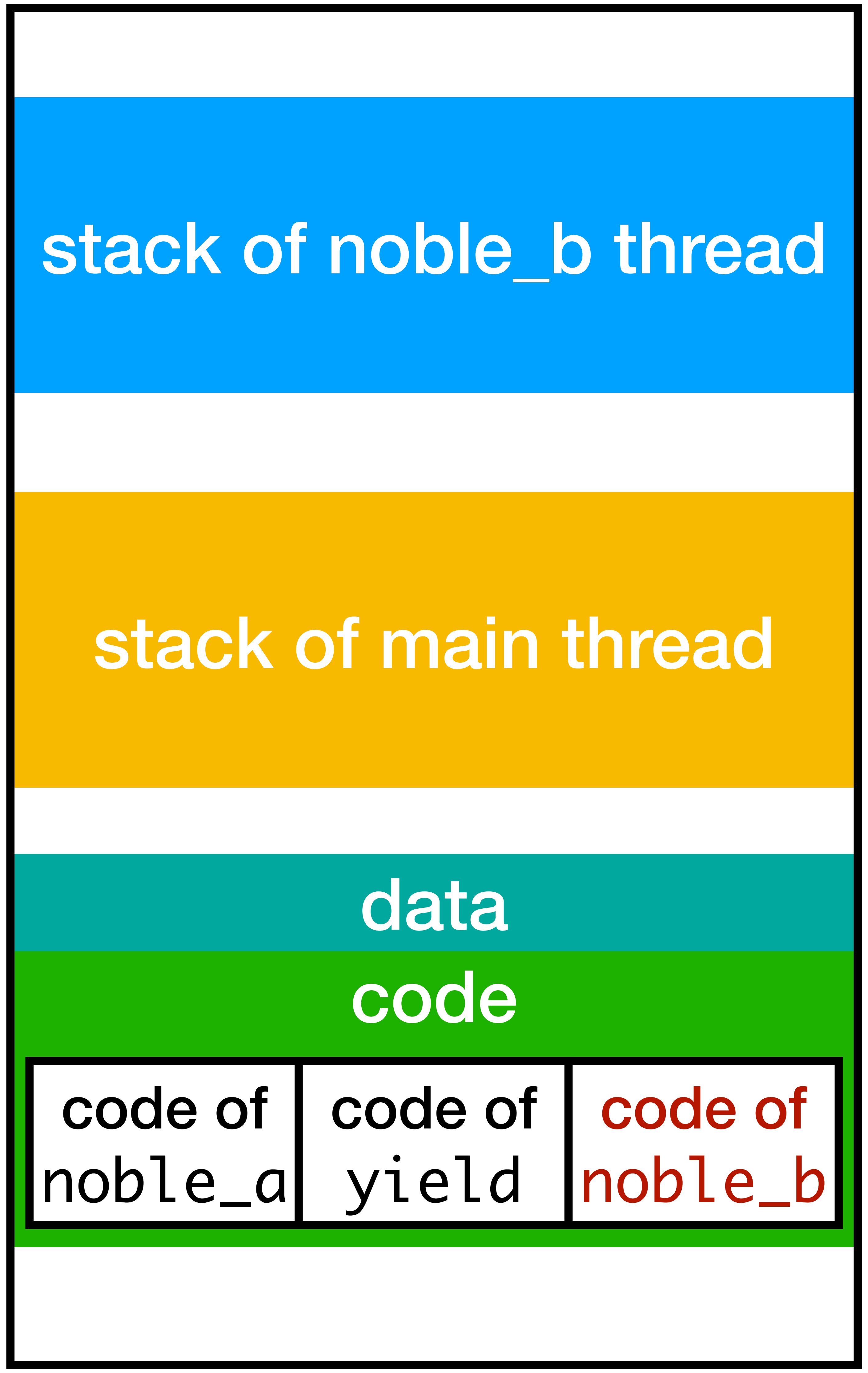


# thread_create

```
struct thread current_thread;
struct queue_t runnable_threads;
……

void thread_create(void (*f)(void *),
                   void *arg,
                   unsigned int stack_size){
    // allocate a stack for noble_b
    // modify global variables
    // call ctx_start to run function f
    // ctx_start is defined in /src/lib/*.s
}

……
```

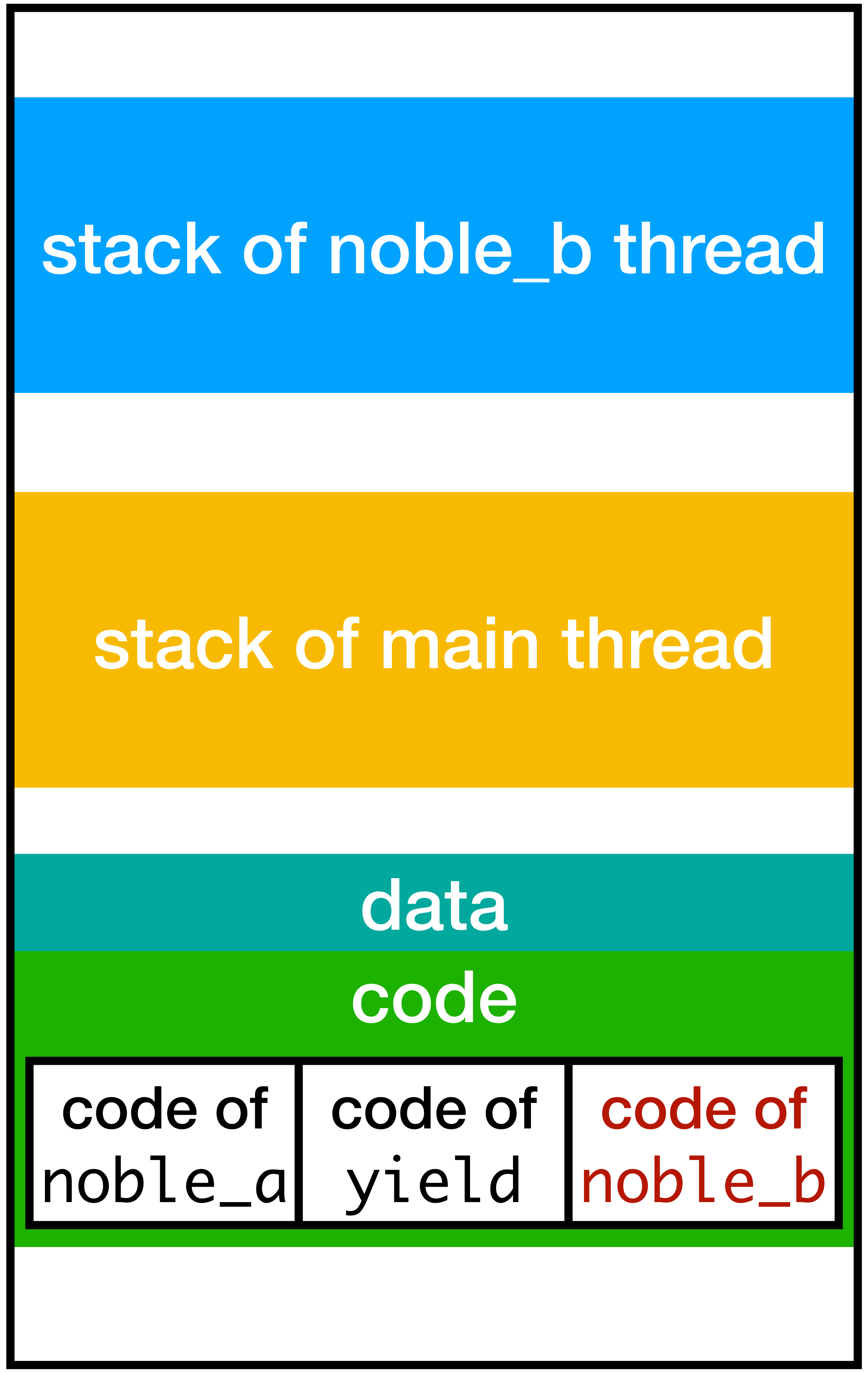


# thread_yield

```
struct thread current_thread;
struct queue_t runnable_threads;
……

void thread_yield(){
    // choose next thread to run
    // call ctx_switch to run the next thread
    // ctx_switch requires previously stored
    // stack pointers
    // ctx_switch is defined in /src/lib/*.s
}

……
```



# Goal of Today's Class

✅ Understand the concepts of context, context-switch and threads

✅ Understand the related functions in assignment P1

- `thread_init`, `thread_create`, `thread_yield`, `thread_exit`
- `ctx_entry`, `ctx_start`, `ctx_switch`
- It's your job to explore the details of how to implement and use these functions. ;-) Try to understand yourself before coming to office hours.

# Homework

- P1 is due on Oct 2. Start early.
- Implement the concepts of thread, context-switch and synchronization of threads (next lecture).