Today’s agenda

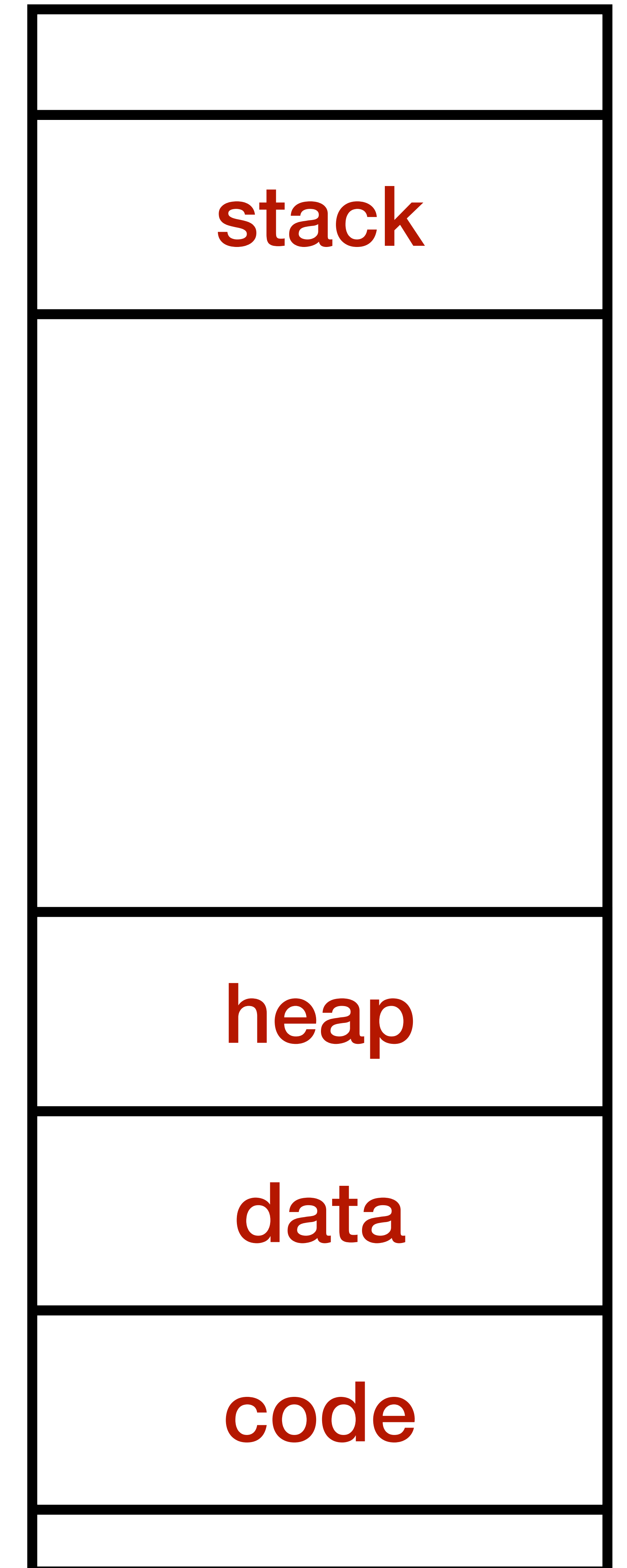
- Review the threading package in P1
- Introduce semaphores
- Demo of EGOS (the Earth and Grass Operating System)

A simple example

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

static void test_code(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}
```

Memory before execution

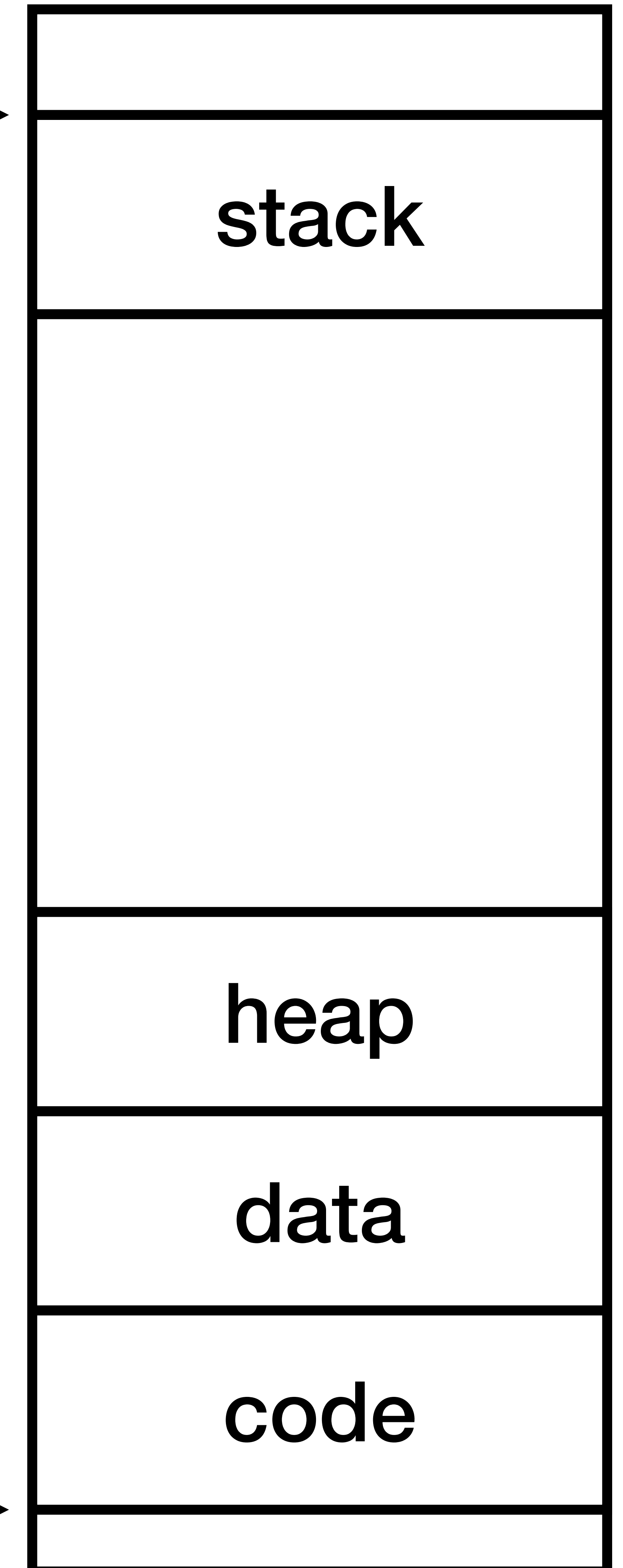
```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

static void test_code(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}
```



CPU before execution

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

static void test_code(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}
```

stack pointer

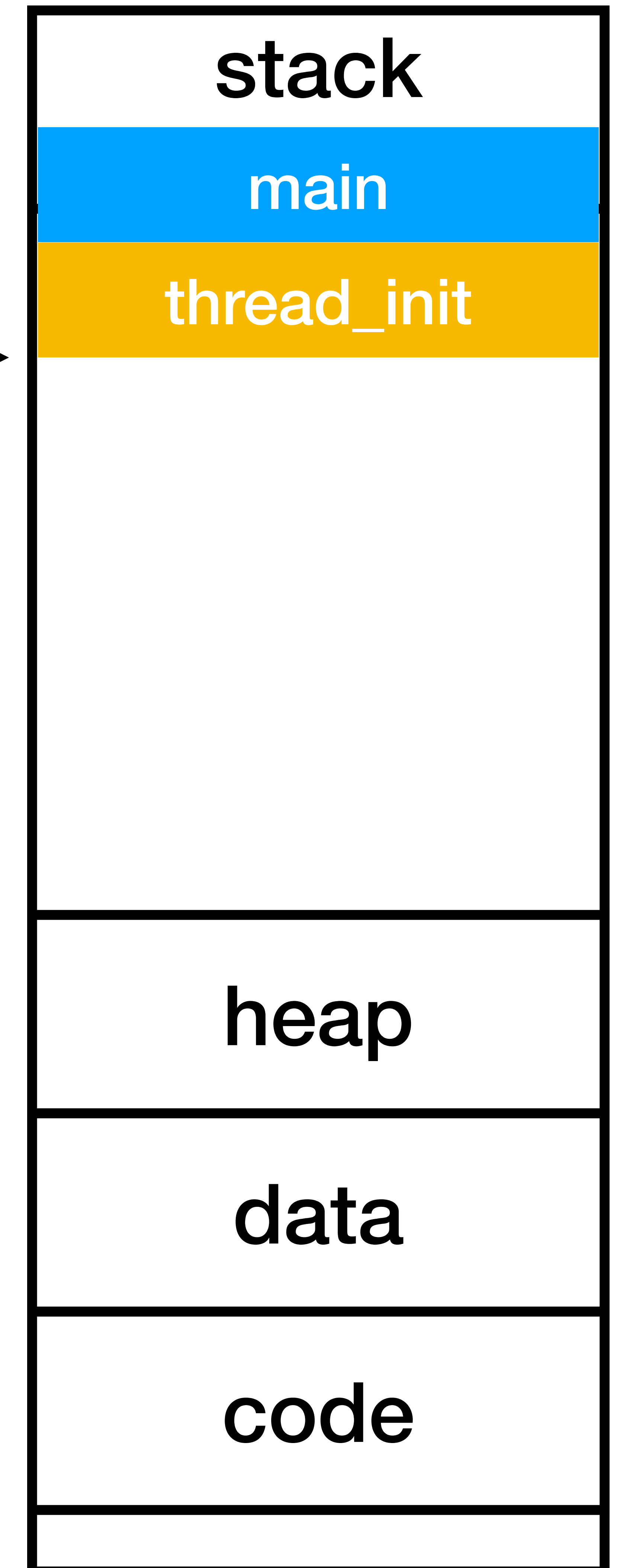


instruction pointer

Initialize threading package

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}
```

stack pointer

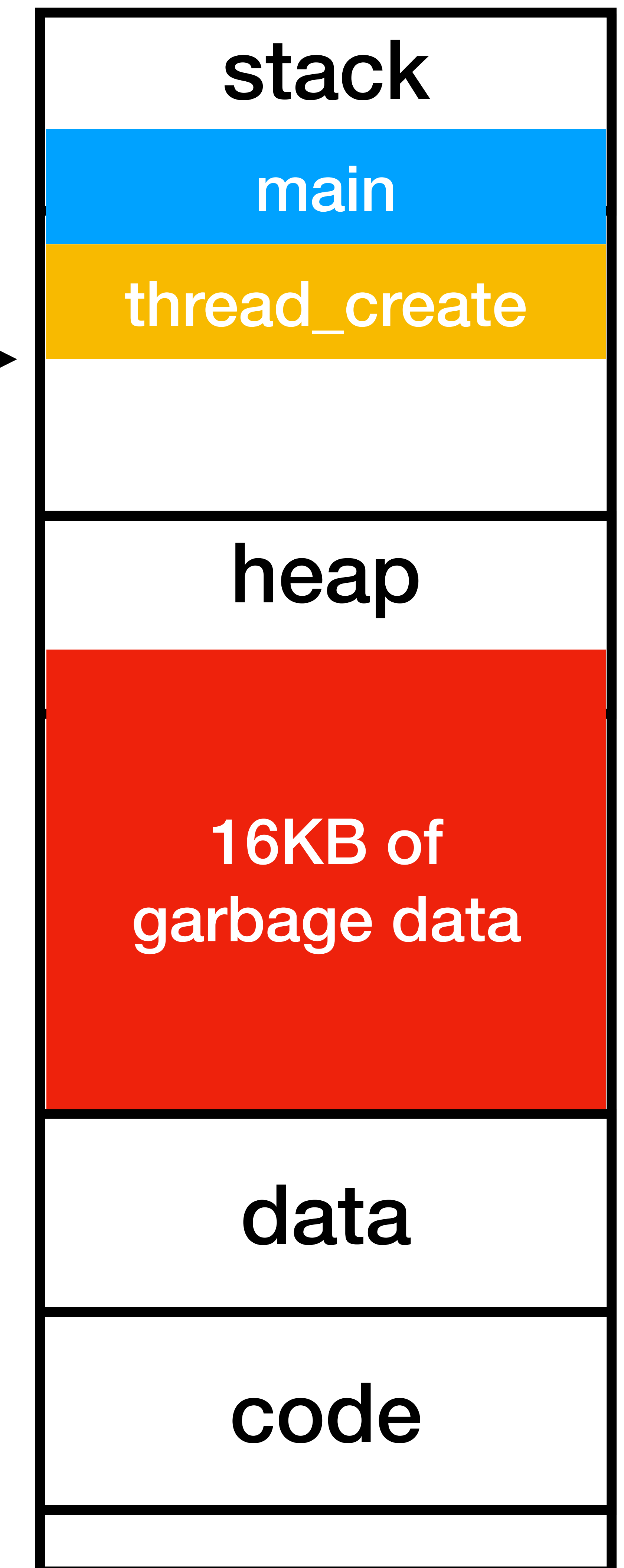


thread_init modifies data and potentially heap

Create a thread, step#1

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}
```

stack pointer



Step #1
thread_create allocates
16KB of memory on heap

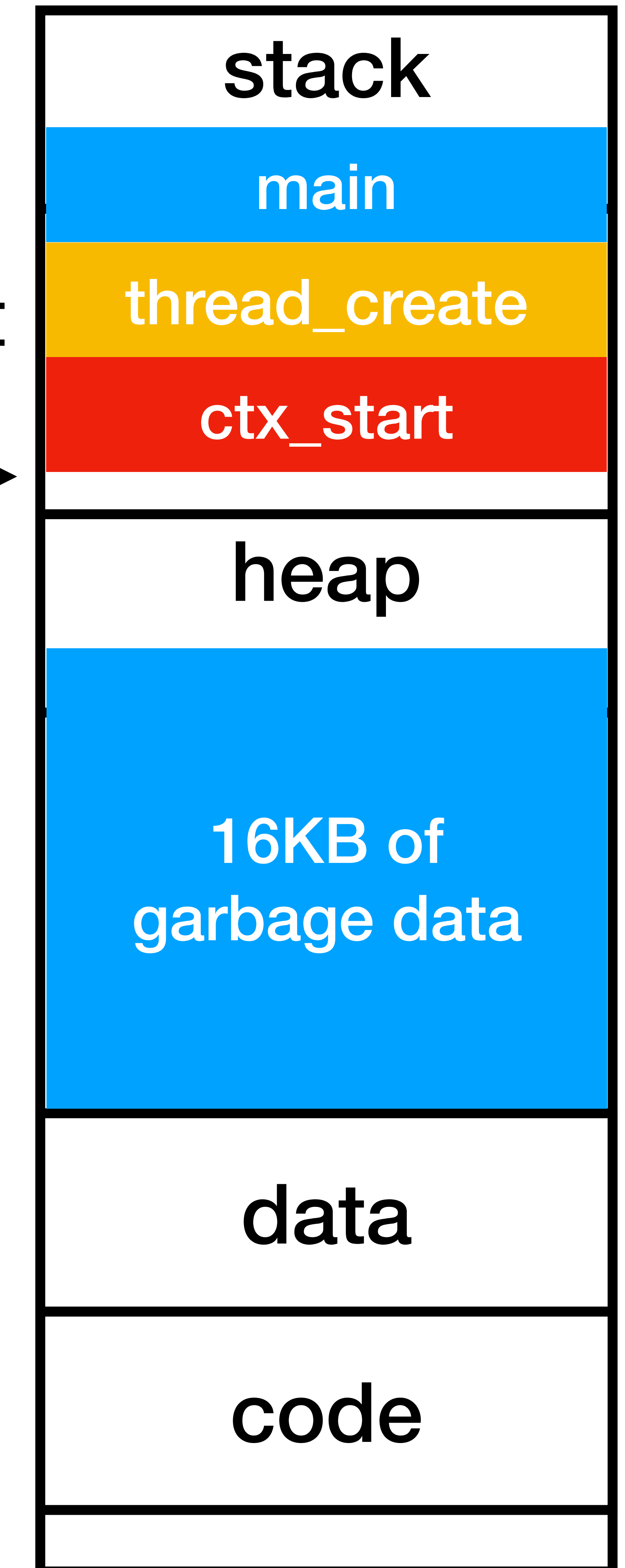
Create a thread, step#2

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

ctx_start:
    …
    movq %rsi, %rsp
    callq ctx_entry
```

Step #2

thread_create calls ctx_start

stack pointer →

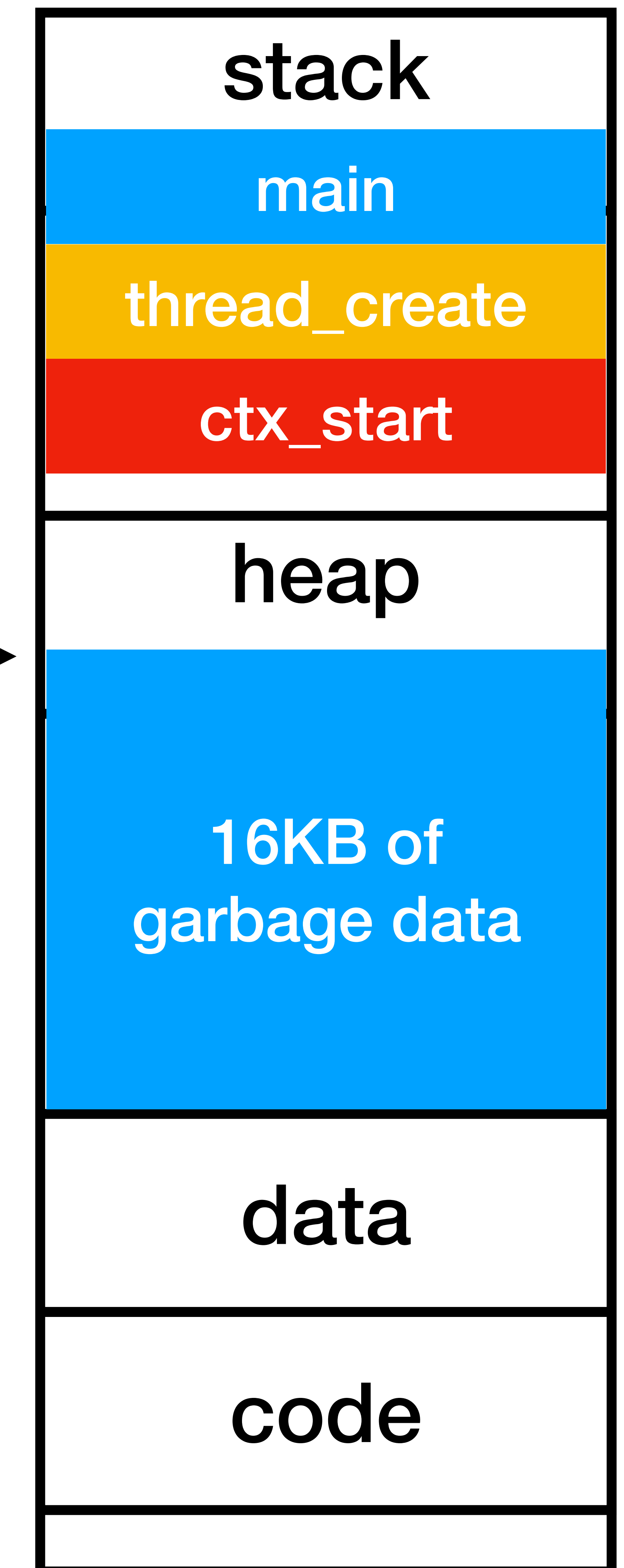


Create a thread, step#3

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

ctx_start:
    …
    movq %rsi, %rsp
    callq ctx_entry
```

Step #3
ctx_start modifies
stack pointer register

stack pointer



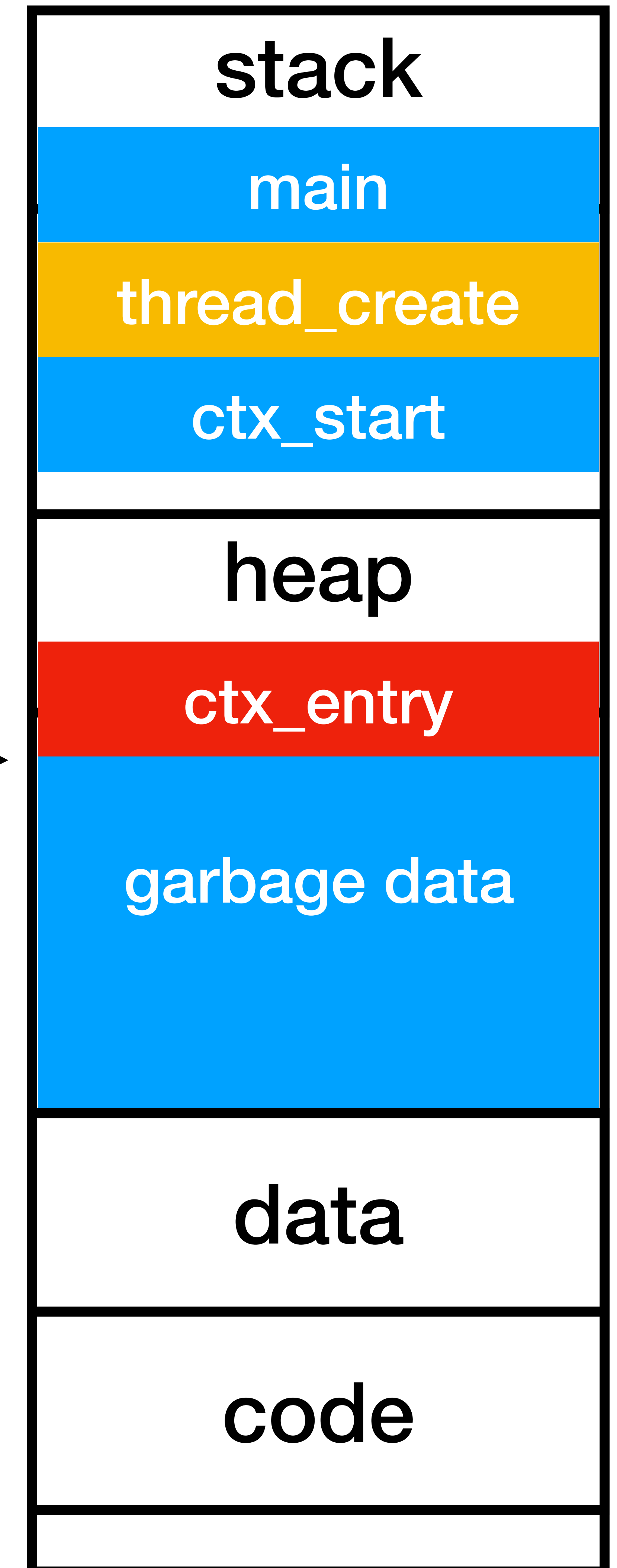
Create a thread, step#4

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}

ctx_start:
    …
    movq %rsi, %rsp
    callq ctx_entry
```

Step #4

ctx_start calls ctx_entry

stack pointer



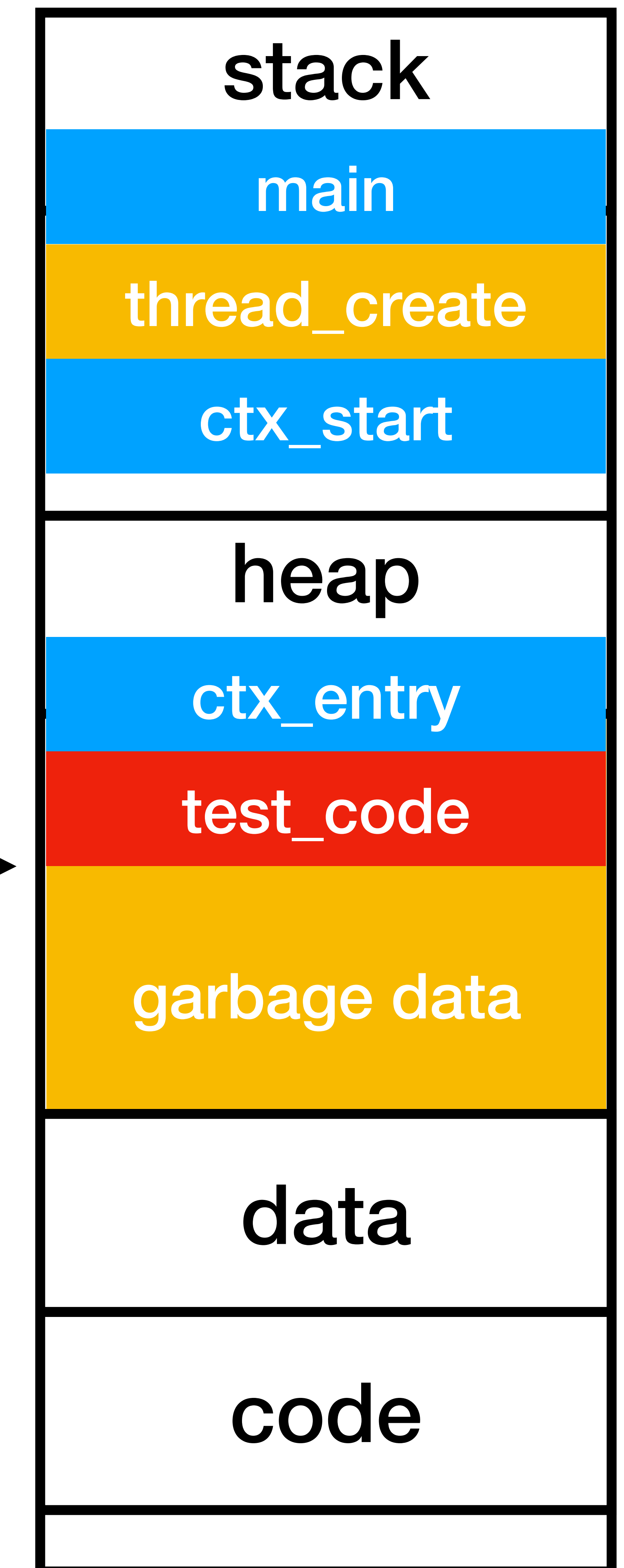
Create a thread, step#5

```
void ctx_entry() {
     // your code here
}

static void test_code(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}
```

Step #5

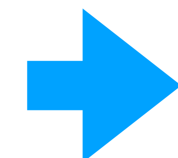
ctx_entry calls test_code

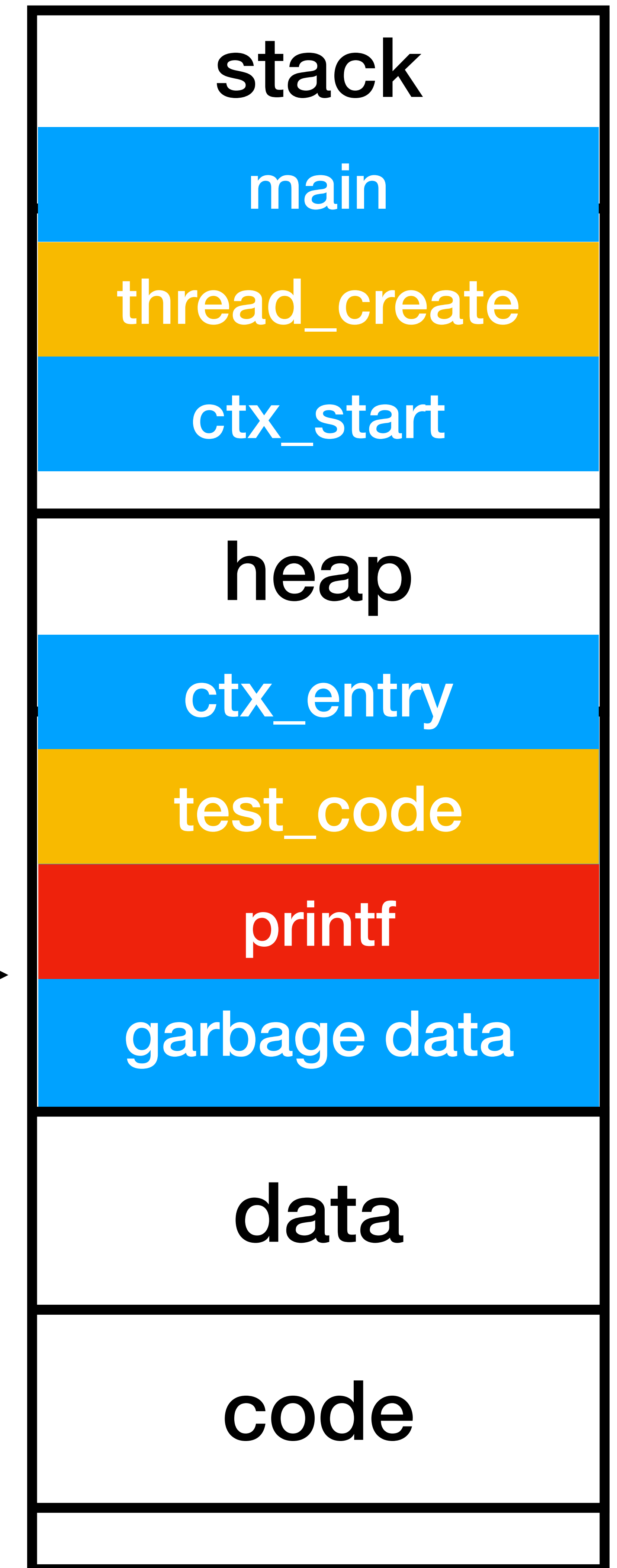
stack pointer



Thread1 executes

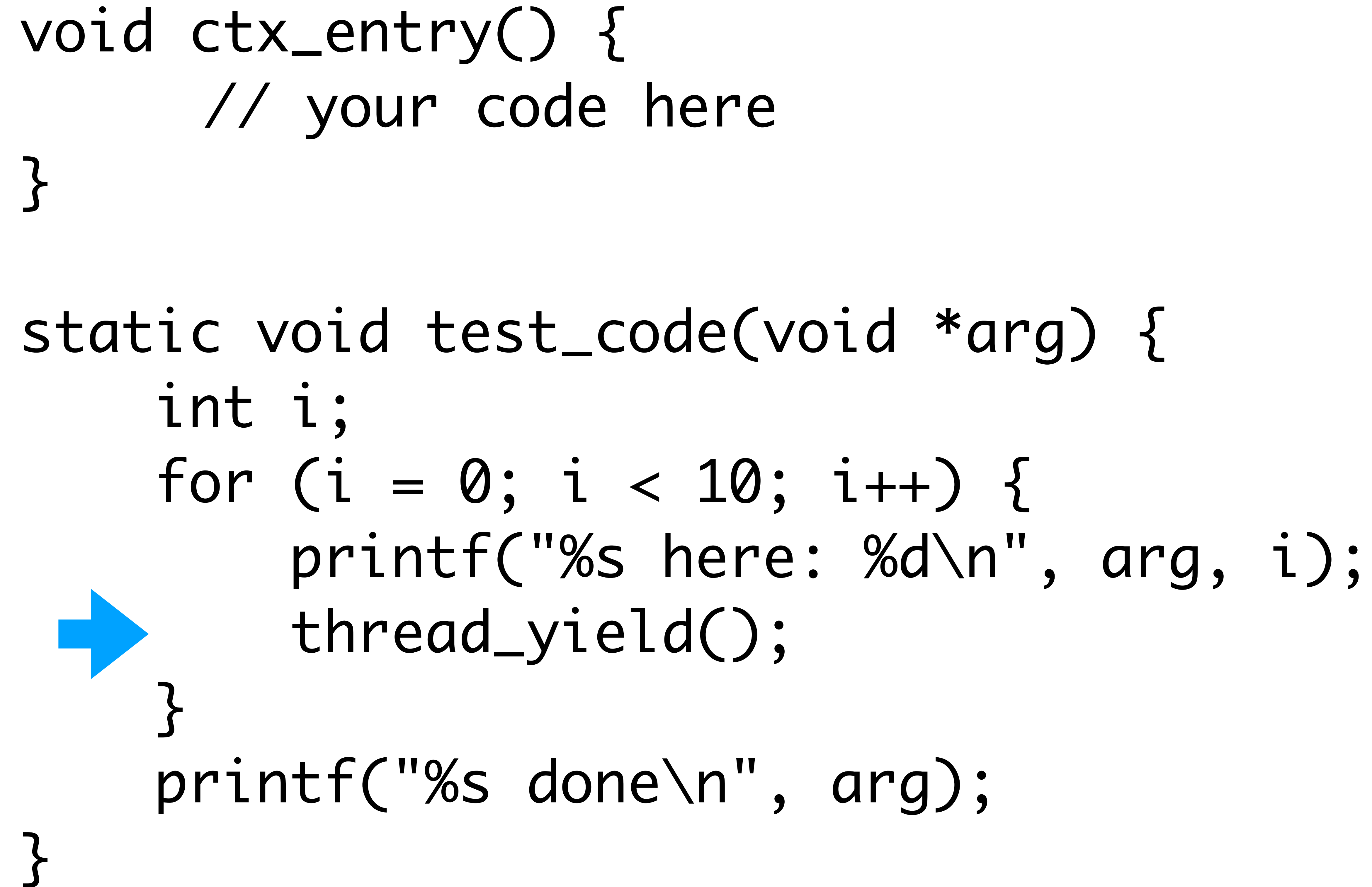
```
void ctx_entry() {
     // your code here
}

static void test_code(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}
```



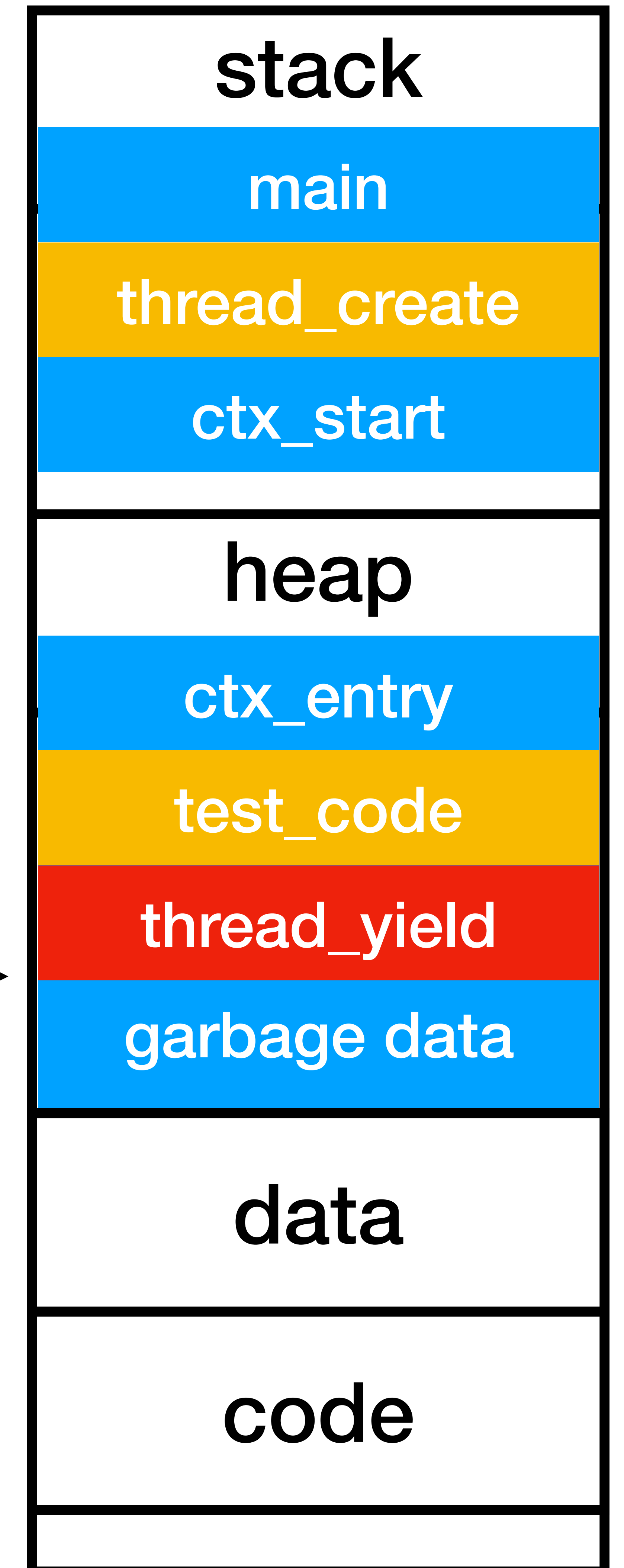


stack pointer

Thread1 yields



stack pointer



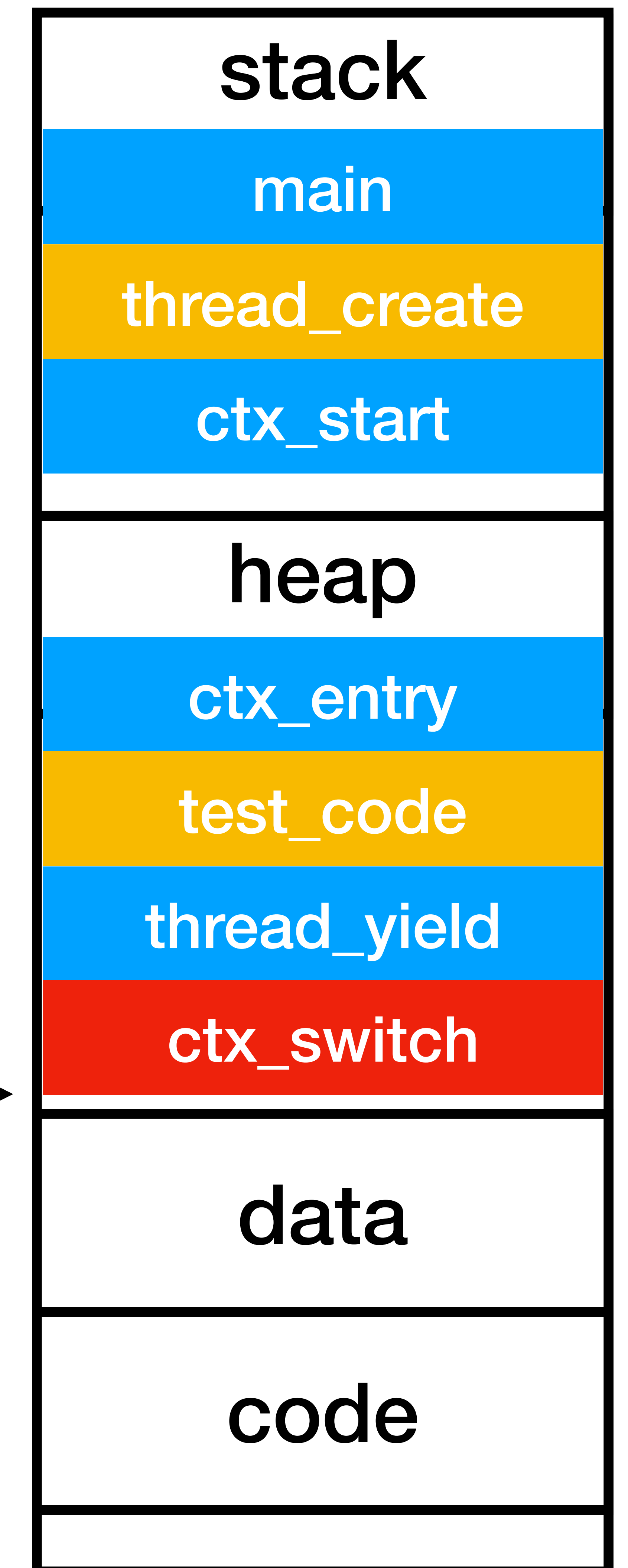
Thread1 yields, step#1

```
void thread_yield() {
    // your code here
}
```

```
ctx_switch:
    …
    movq %rsi, %rsp
    …
    retq
```

Step #1
thread_yield calls
ctx_switch

stack pointer



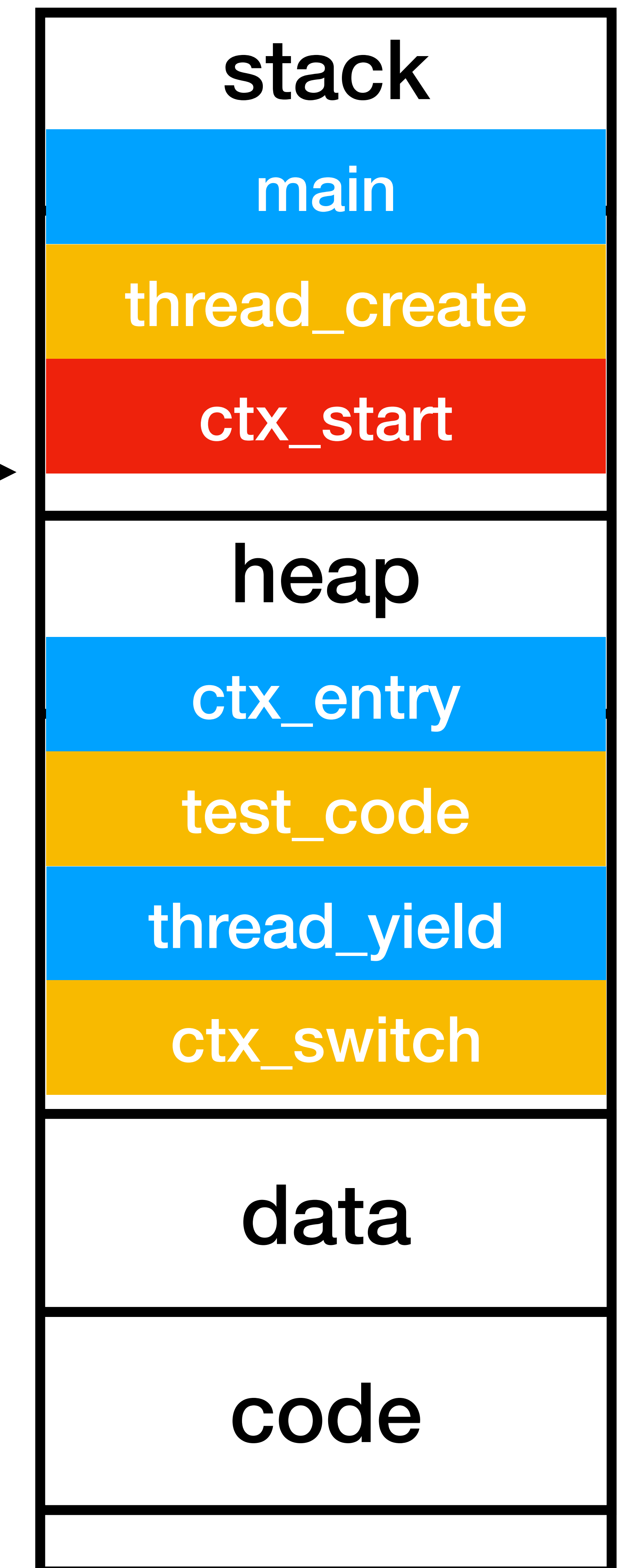
Thread1 yields, step#2

```
void thread_yield() {
    // your code here
}
```

```
ctx_switch:
    …
    movq %rsi, %rsp
    …
    retq
```

Step #2
ctx_switch modifies
stack pointer register

stack pointer

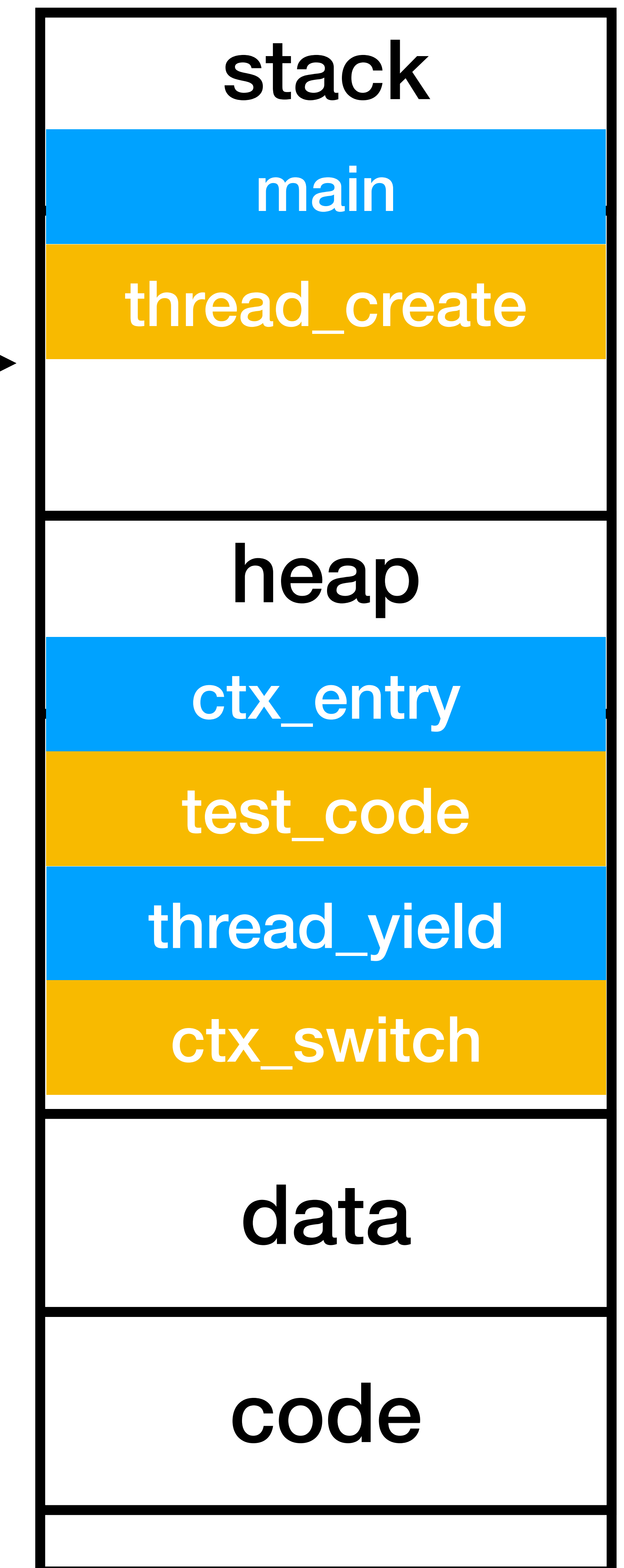


Thread1 yields, step#3

```
void thread_yield() {
    // your code here
}


ctx_switch:
    …
    movq %rsi, %rsp
    …
    retq
```

Step #3
ctx_switch executes the return instruction

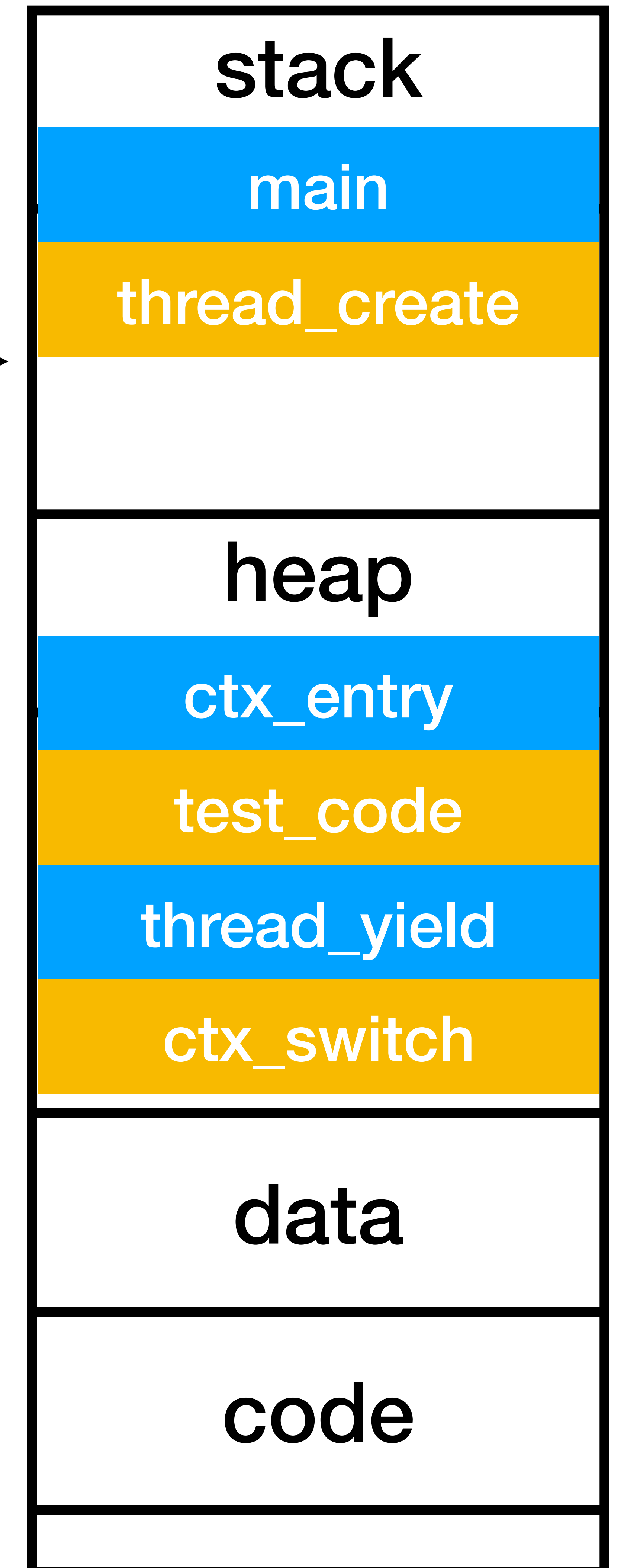
stack pointer



Main thread continues

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}
```

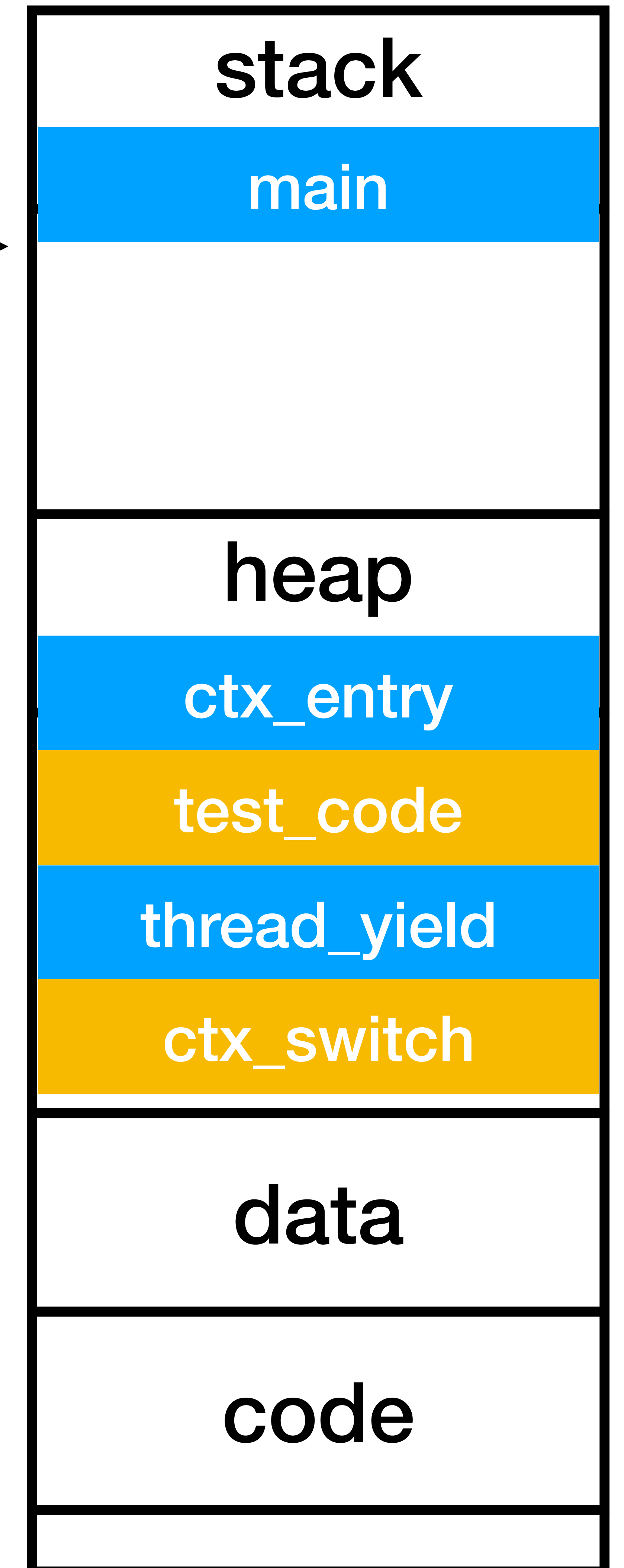
stack pointer



Main thread continues

```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
    thread_exit();
    return 0;
}
```

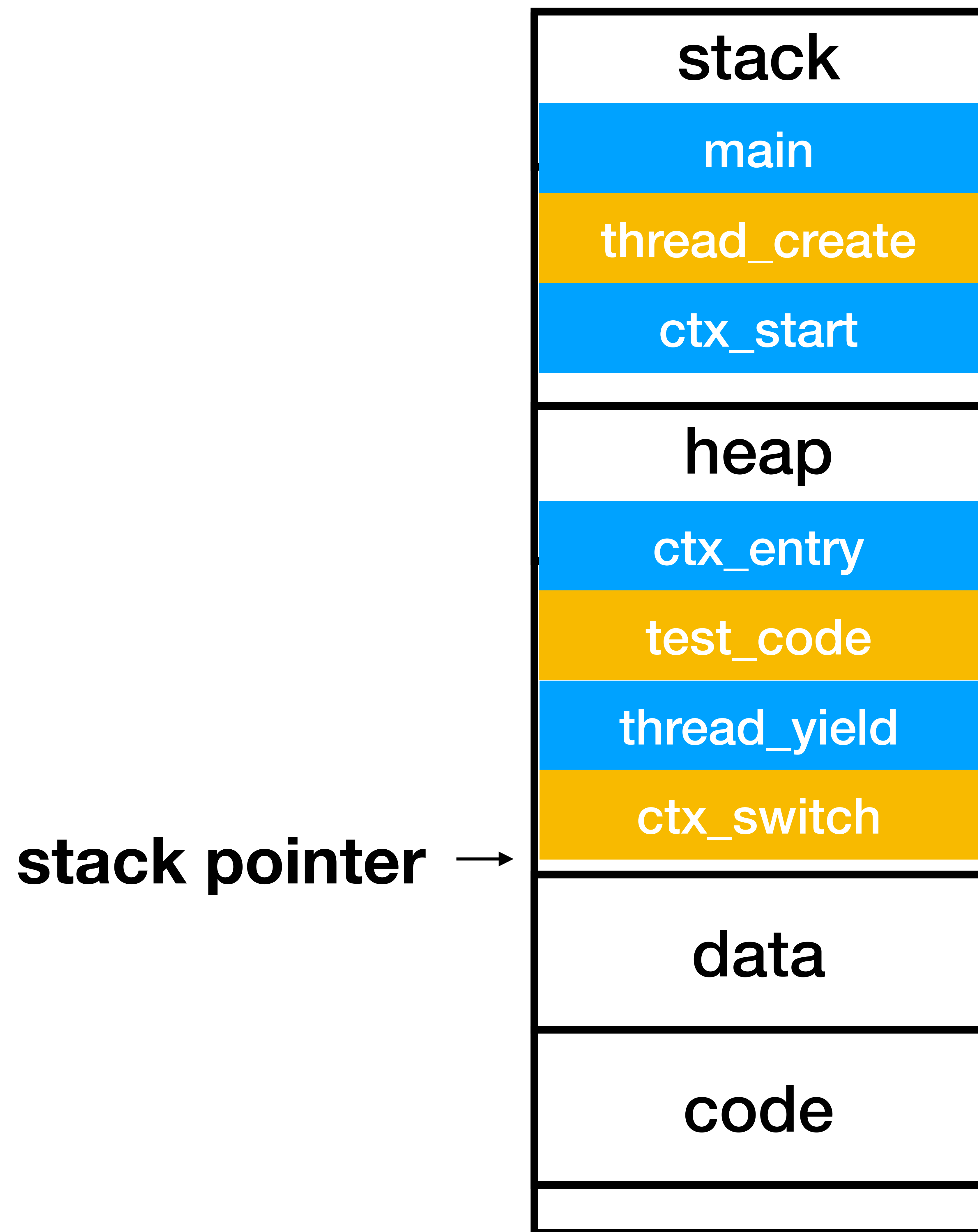
stack pointer



Recall the lesson in week2

context = memory address space
+ stack pointer + instruction pointer

Before the `movq` instruction

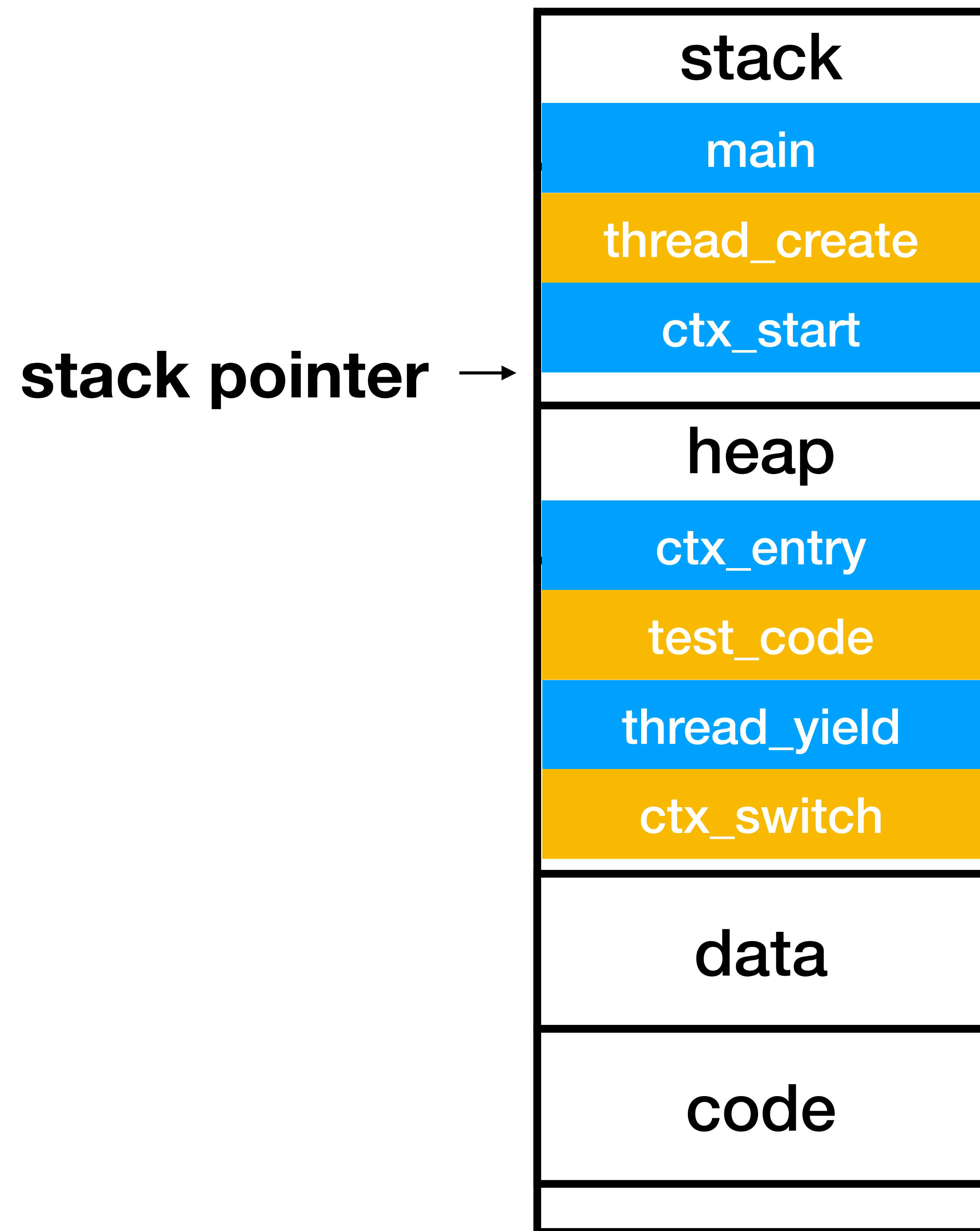


instruction pointer →

```
ctx_switch:
  …
  movq %rsi, %rsp
  …
  retq
```

CPU is in the **context** of **thread1**

After the `movq` instruction

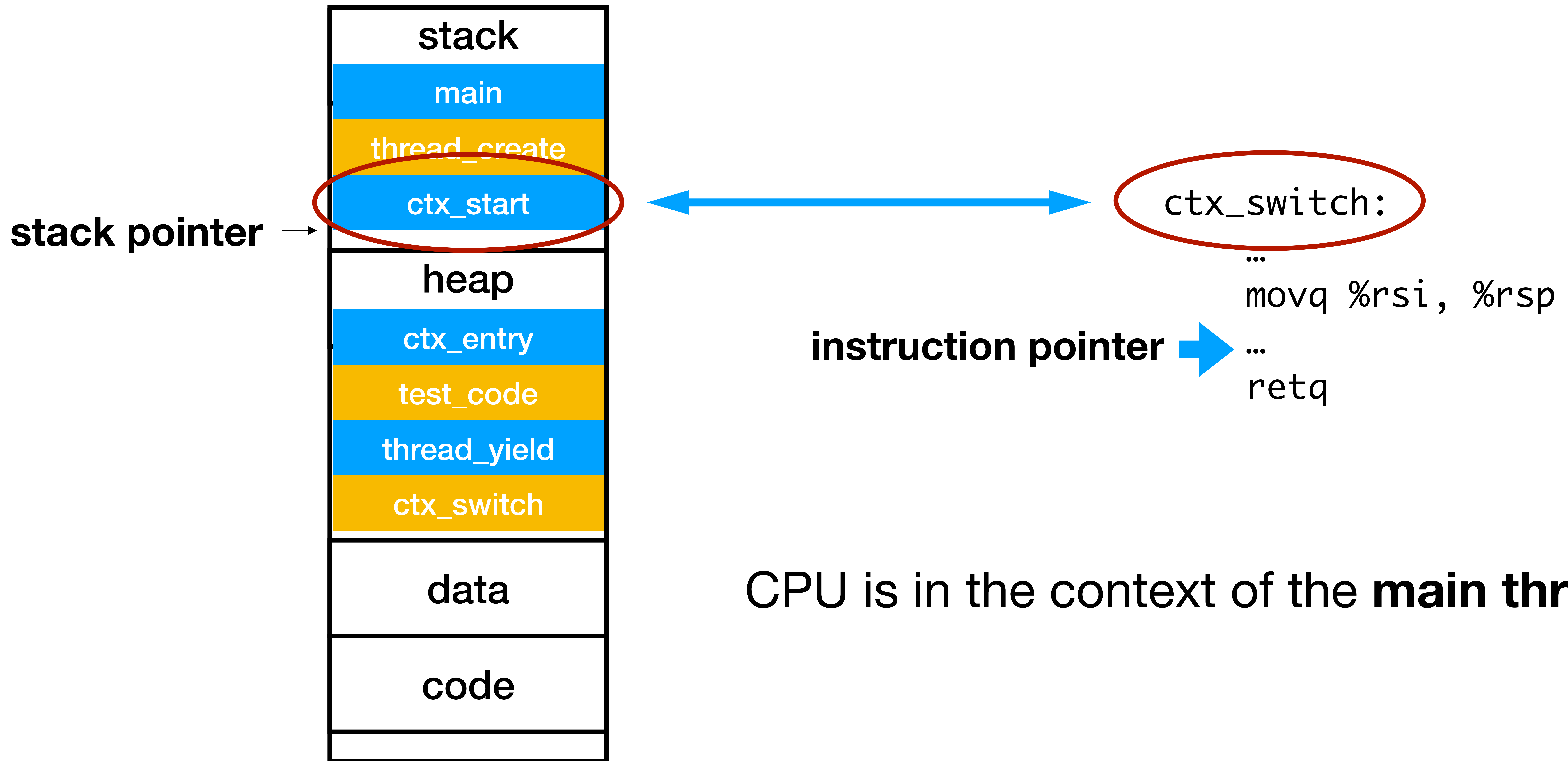



```
ctx_switch:
    …
    movq %rsi, %rsp
    …
    retq
```

instruction pointer → (at the `…` after `movq`)

CPU is in the **context** of the **main thread**

Why ctx_switch can use the stack of ctx_start ?



CPU is in the context of the **main thread**

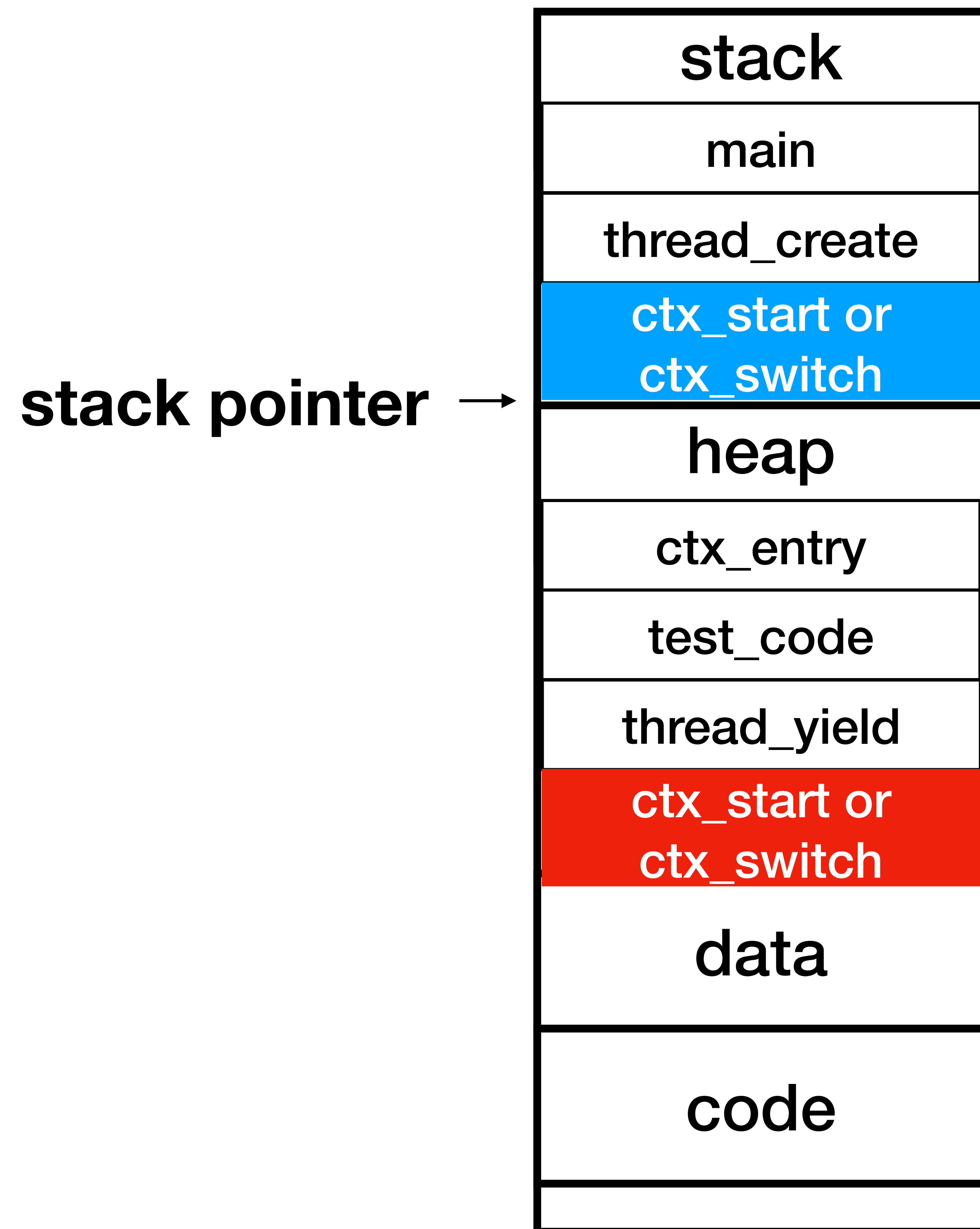
Why `ctx_switch` can use the stack of `ctx_start` ?

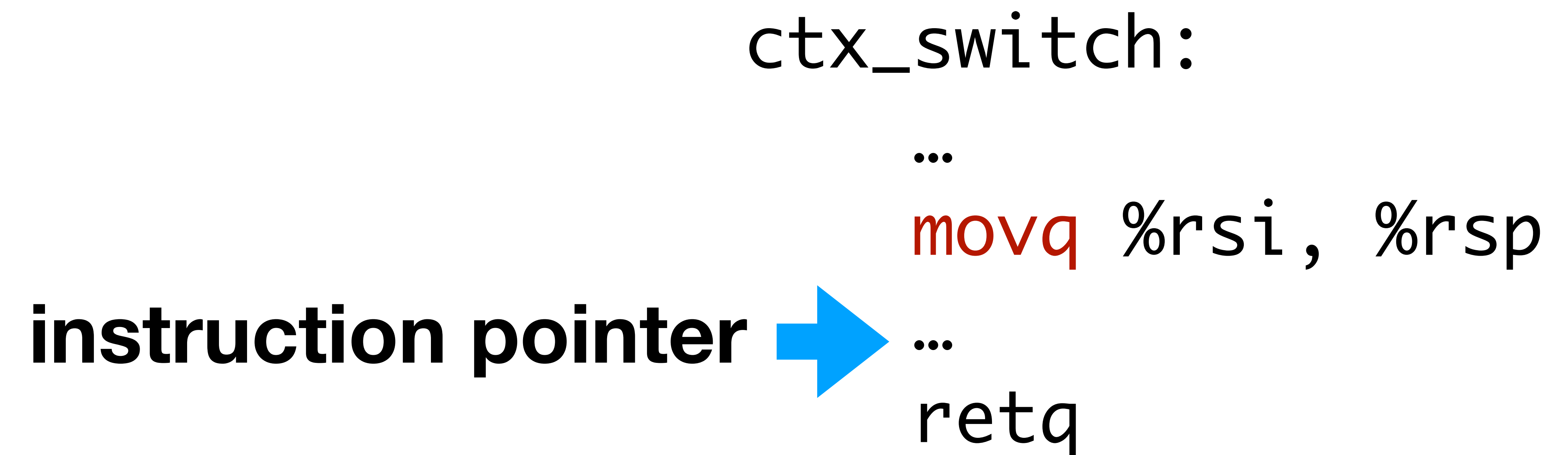
```
ctx_switch:  // ip already pushed!
    pushq    %rbp
    pushq    %rbx
    pushq  %r15
    pushq  %r14
    pushq  %r13
    pushq  %r12
    pushq  %r11
    pushq  %r10
    pushq  %r9
    pushq  %r8
    movq   %rsp,  (%rdi)
    movq    %rsi,  %rsp
    popq   %r8
    popq   %r9
    popq   %r10
    popq   %r11
    popq   %r12
    popq   %r13
    popq   %r14
    popq   %r15
    popq      %rbx
    popq      %rbp
    retq
```

```
ctx_start:
    pushq  %rbp
    pushq  %rbx
    pushq  %r15
    pushq  %r14
    pushq  %r13
    pushq  %r12
    pushq  %r11
    pushq  %r10
    pushq  %r9
    pushq  %r8
    movq   %rsp, (%rdi)
    movq   %rsi, %rsp
    callq  ctx_entry
```

The two functions have the same stack structure: they push the same registers in the same order

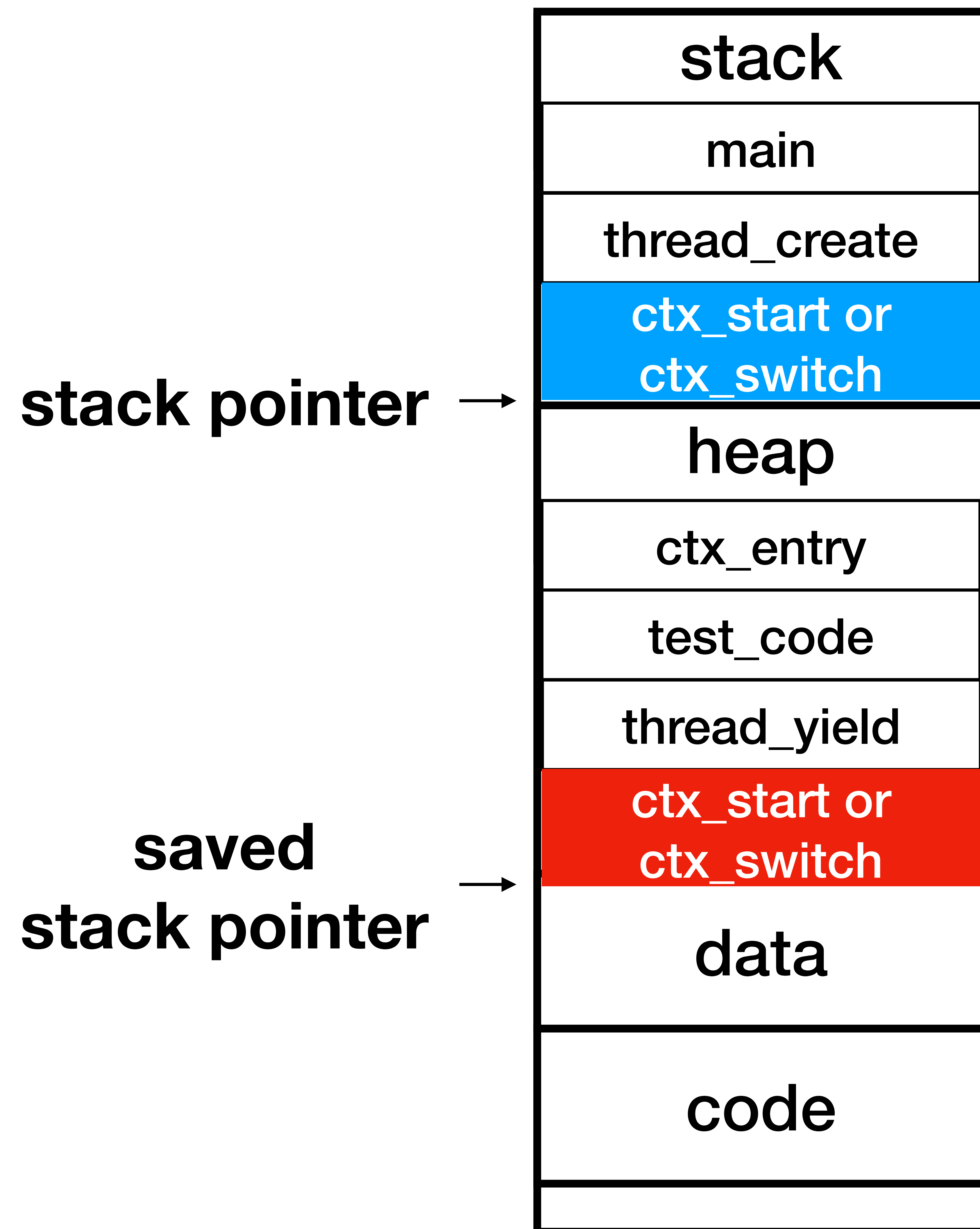
View the frame as `ctx_start` or `ctx_switch`





CPU is in the **context** of the **main thread**

Why saving stack pointer in `struct thread`?




```
ctx_switch(&current->sp, next->sp)

// save the address of the ctx_start/
ctx_switch frame of thread1

// set the stack pointer to the
ctx_start/ctx_switch frame of main thread
```

Every runnable thread has a ctx_start/ctx_switch frame at its stack top so that it can switch back

Question

Why the definition never mentions registers?

context = memory address space
+ stack pointer + instruction pointer

Today's agenda

- Review the threading package in P1
- ➡ Introduce semaphores
- Demo of EGOS (the Earth and Grass Operating System)

Why semaphores?

- We’re not teaching general semaphores in CS4410 anymore
- But it is a simple way of testing your threading package

Semaphores as counters

```
// Initialize a counter and set to count
void sema_init(struct sema *sema, unsigned int count);

// Increment the counter by 1
void sema_inc(struct sema *sema);

// Wait until counter > 0, then decrement counter by 1
void sema_dec(struct sema *sema);

// Release the counter
bool sema_release(struct sema *sema);
```

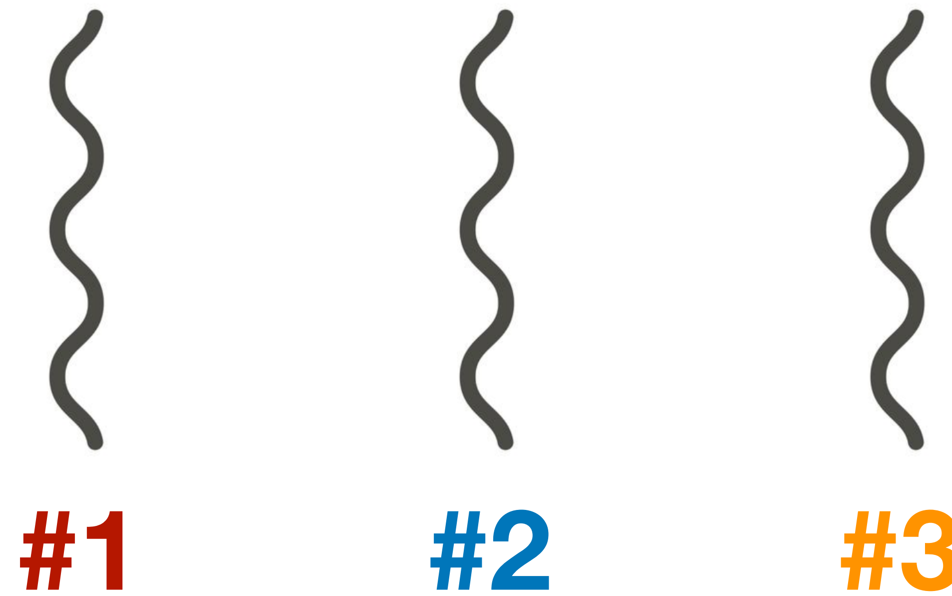
Example: bounded buffer producer-consumer

A fixed length array as a buffer holding items
between producer and consumer threads

Index	#1	#2	#3
Content			

Example: bounded buffer producer-consumer

producer threads



Index	#1	#2	#3
Content	item from **#1**	item from **#1**	item from **#2**

Now the buffer is full
All producer threads need to wait

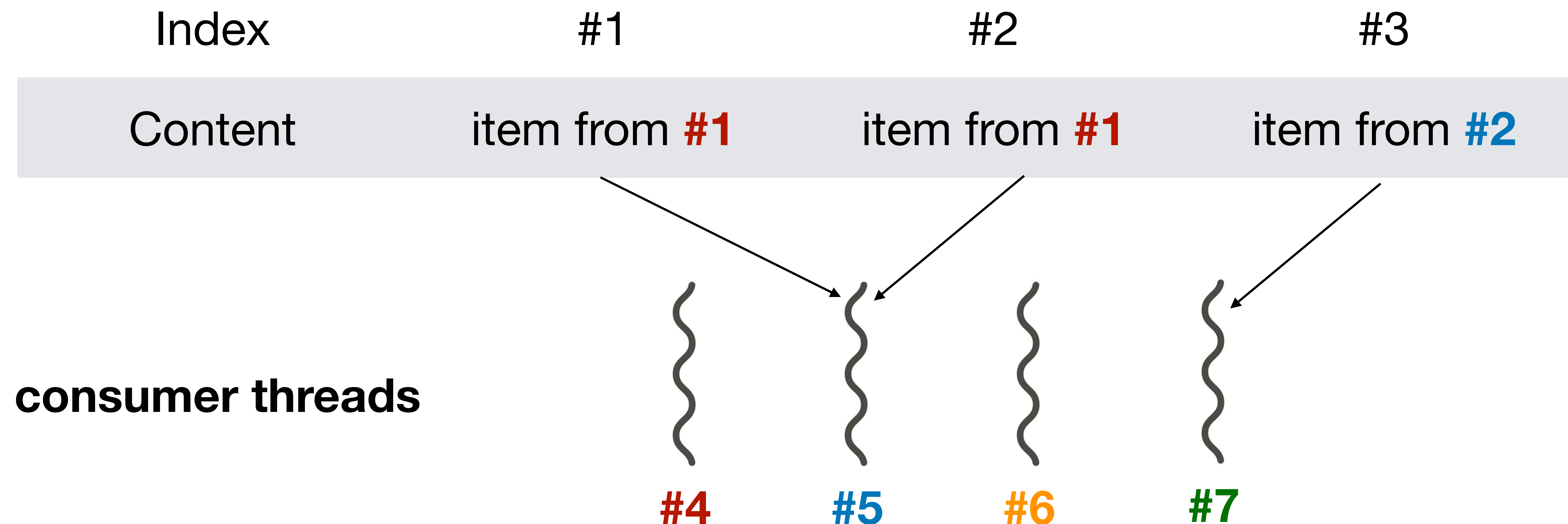
Example: bounded buffer producer-consumer

Say,

thread **#5** consumes the first two items

thread **#7** consumes the third item

Now the buffer is empty, all consumer threads need to wait



Example: bounded buffer producer-consumer

- Figure 2 in P1 handout
- Hint: each semaphore maintains a queue of waiting threads
  - When a thread tries to decrement a zero semaphore, put its `struct thread` onto this waiting queue and yield
  - When incrementing a zero semaphore, dequeue a thread if the waiting queue is non-empty; otherwise, set counter to 1
- In P1, you need to implement similar synchronization puzzles with semaphores, such as dining philosophers, reader/writer lock, …

Today's agenda

- Review the threading package in P1
- Introduce semaphores
- ➔ Demo of EGOS (the Earth and Grass Operating System)

EGOS overview

- Earth: hardware-specific layer
  - disk, screen, keyboard drivers
  - interrupt handler, memory protection, etc.
- Grass: hardware-independent layer
  - process, system call and inter-process communication
- Application layer
  - file system and shell
  - shell commands: ls, mkdir, echo, cat, …