Announcements

- P0 deadline is postponed to Sunday (Sep. 11)
  - with two slip days, it will be Tuesday (Sep. 13)
- Useful Ed posts:
  - `queue_iterate` example: post #8
  - memory leak and `valgrind`: post #25, #31
  - what to submit: post #16; grading: post #32

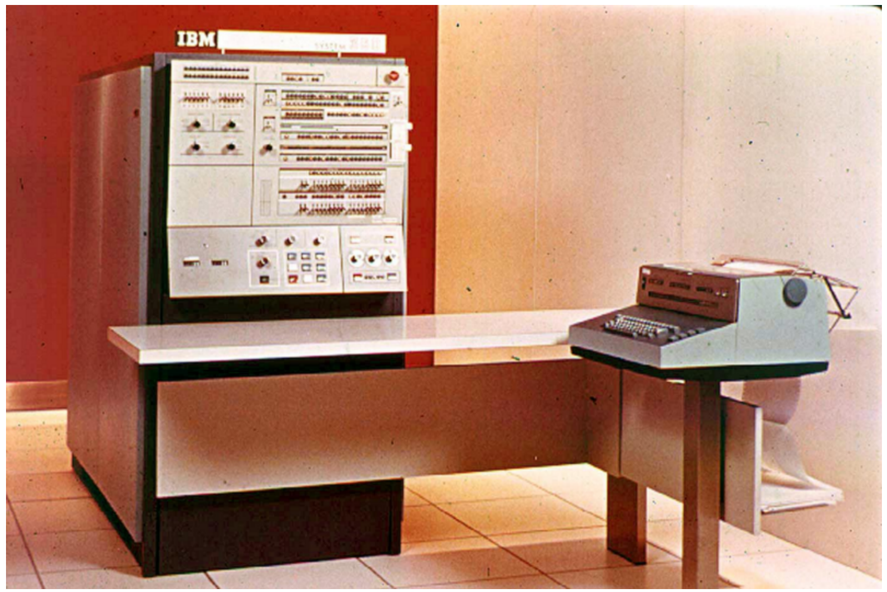
Context and Multi-threading

P1 and P2: Multi-threading

- P1: without CPU support
  - Keywords: context, threads and context-switch
- P2: with CPU support
  - Keywords: timer interrupt, scheduling and priority

Why multi-threading?

faster



Early 1960s:
Human wait for computers.

Late 1960s:
Computers wait for human.

* **Left: IBM 709 https://www.computerhistory.org/collections/catalog/102728984**
* **Right: IBM 360 https://about.sourcegraph.com/blog/the-ibm-system-360-the-first-modular-general-purpose-computer/**

An example of multi-threading

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

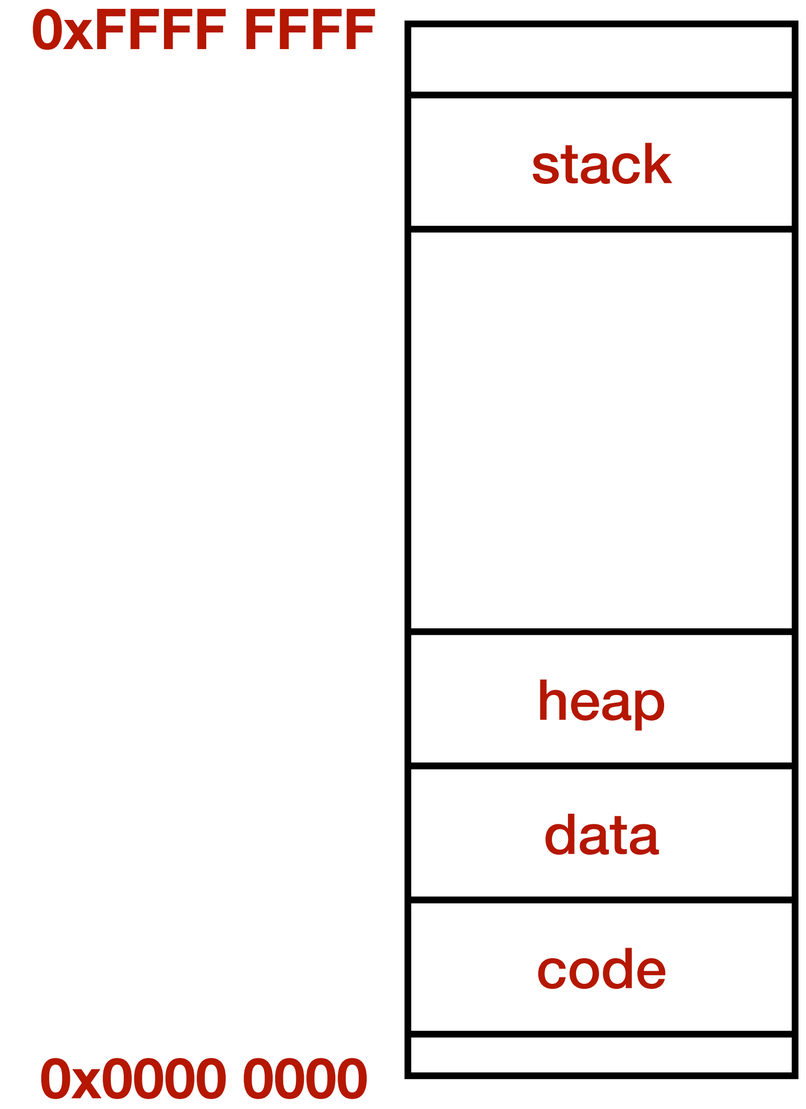
One possible output

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

```
thread1 here: 0
thread2 here: 0
thread1 here: 1
main thread here: 0
thread2 here: 1
thread1 here: 2
main thread here: 1
thread2 here: 2
thread1 done
main thread here: 2
thread2 done
main thread done
```

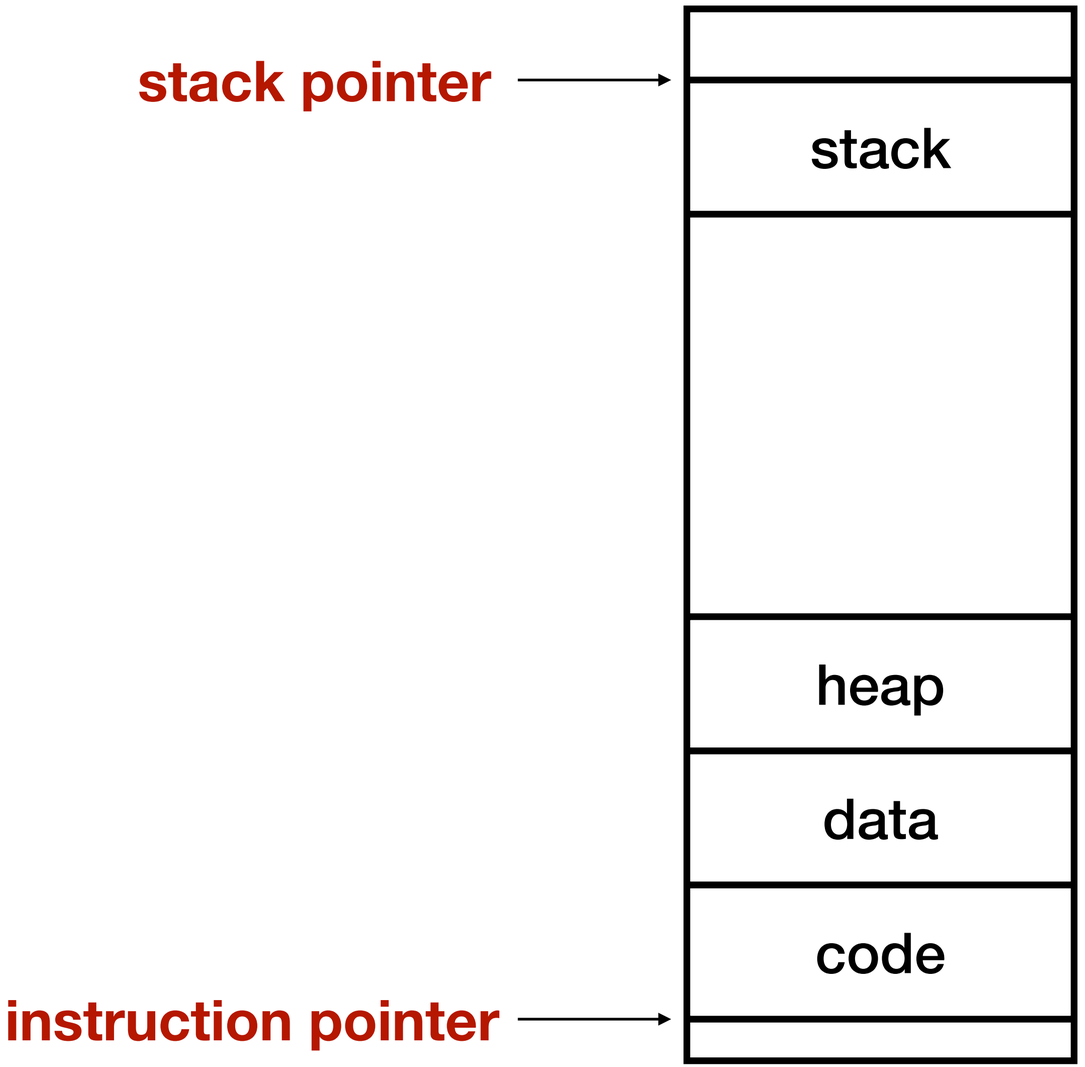
Memory before execution

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```



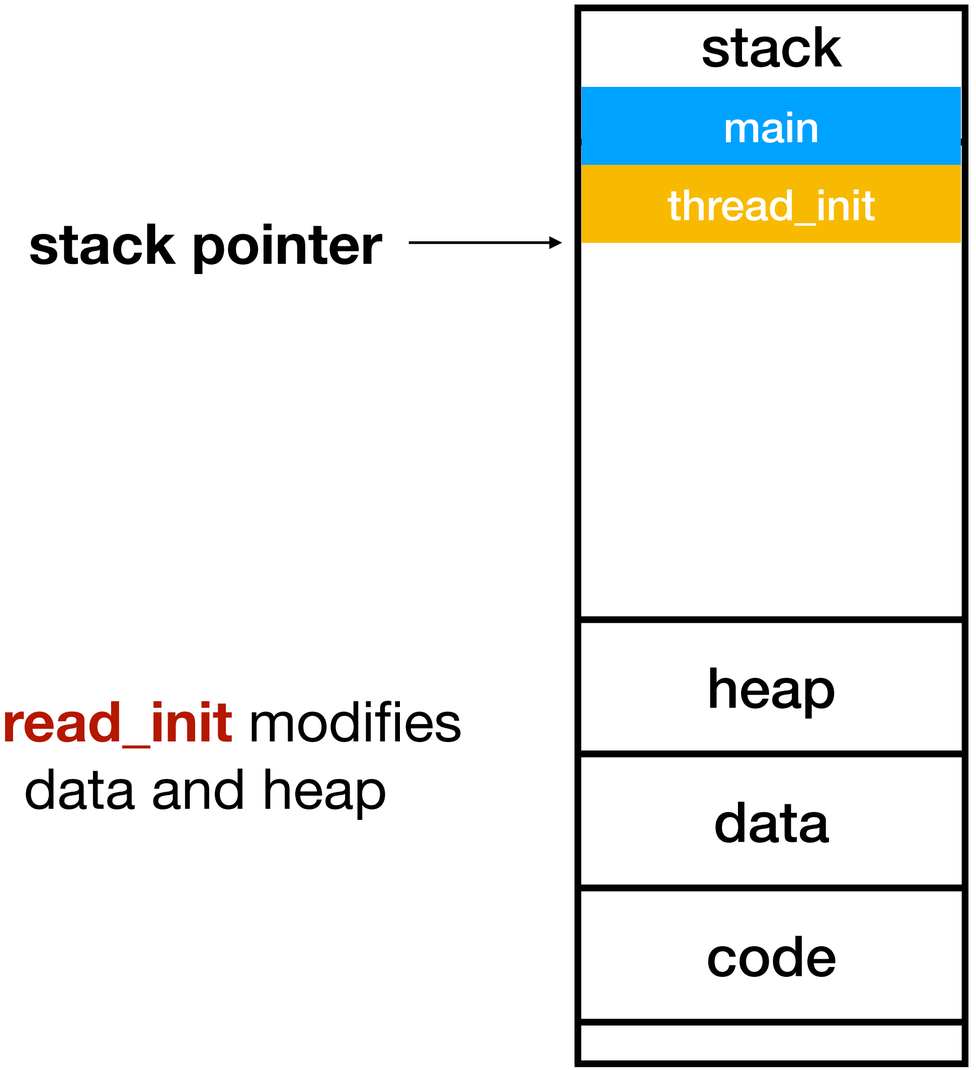
CPU before execution

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```



Initialize some data structures

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

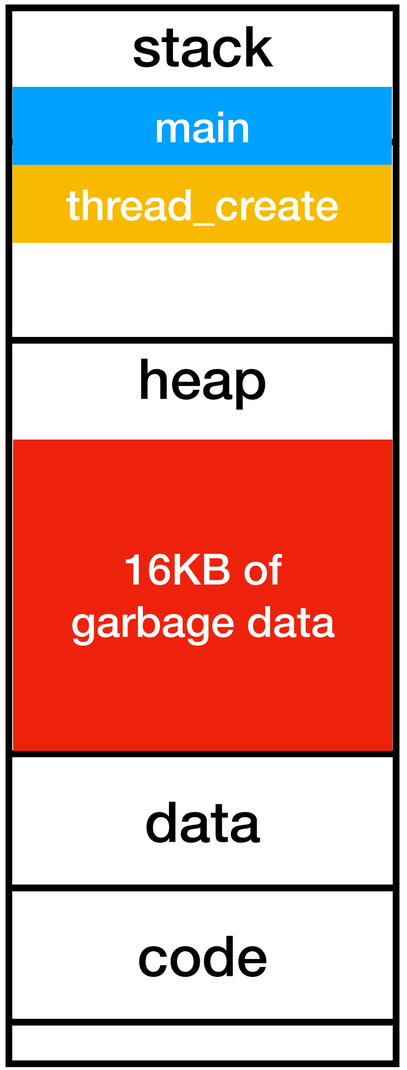


thread_init modifies data and heap

Create a thread, step 1/5

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

stack pointer →

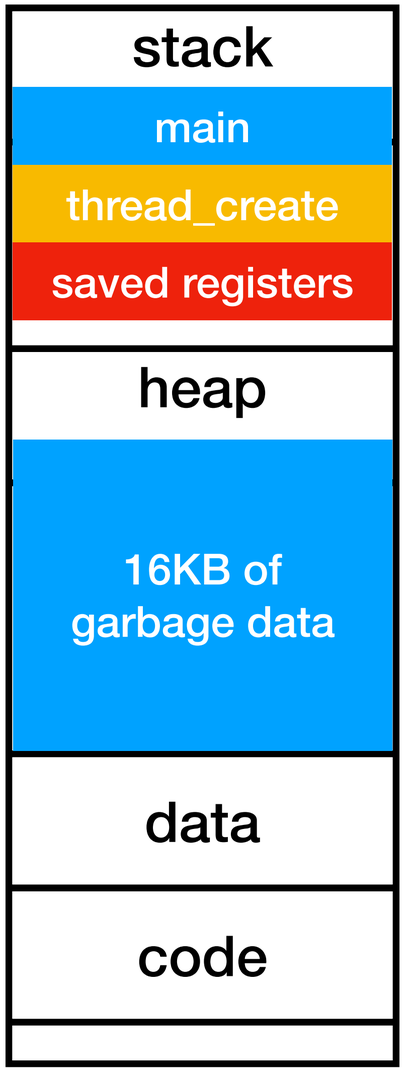


Step 1/5

thread_create allocates
16KB of memory on heap

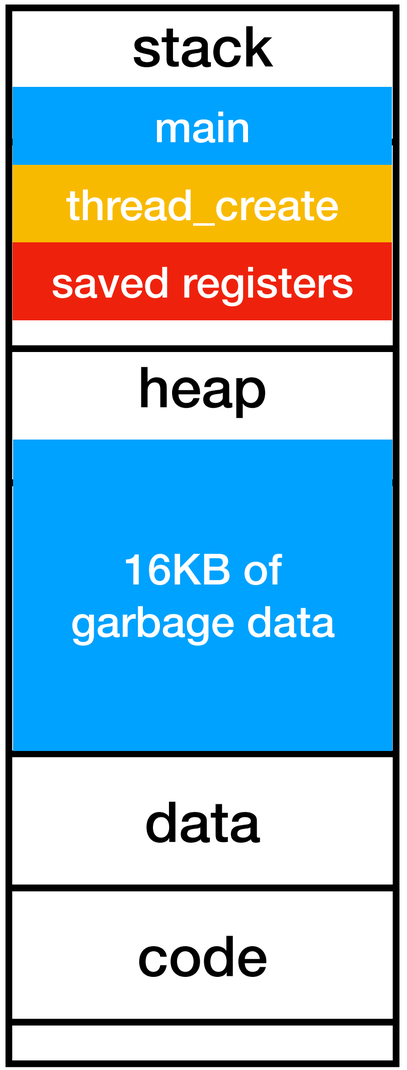
Create a thread, step 2/5

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

ctx_start:  // step 2/5: thread_create() calls ctx_start()
    …       // save registers on the stack with store instructions
    mv sp, a1
    call ctx_entry
```



Create a thread, step 3/5

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

ctx_start:          // step 3/5: thread_create() passes the new
    …               // stack pointer as 2nd argument to ctx_start()
    mv sp, a1       // ctx_start() modifies sp to its 2nd argument
    call ctx_entry
```

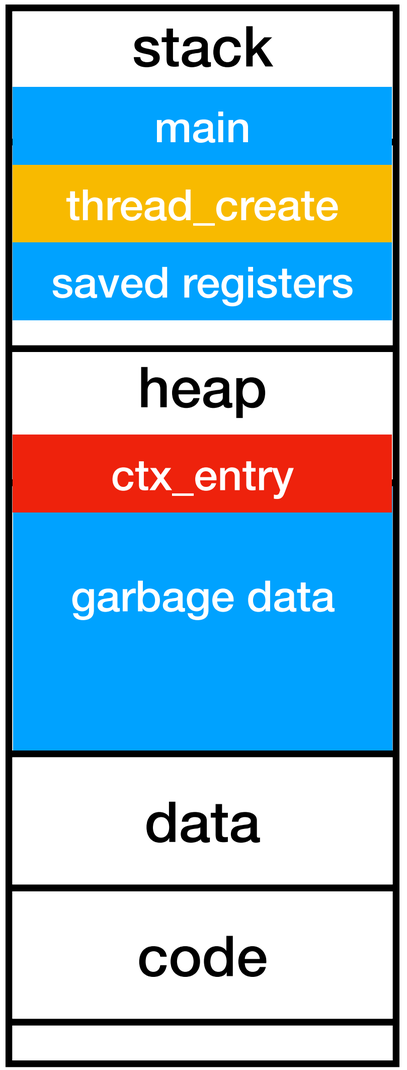


old stack pointer

new stack pointer

Create a thread, step 4/5

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

ctx_start:
    …
    mv sp, a1
    call ctx_entry  // step 4/5: call function ctx_entry()
```



stack pointer →

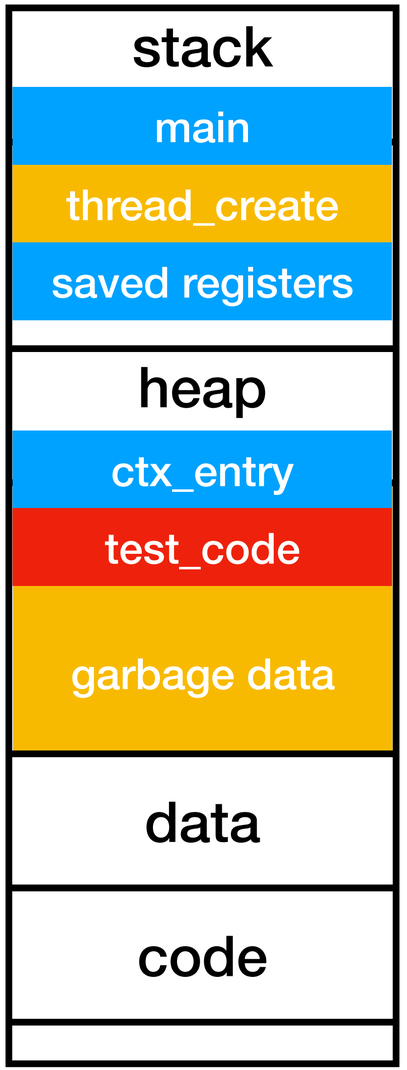
Create a thread, step 5/5

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

Step 5/5

ctx_entry calls **test_code**

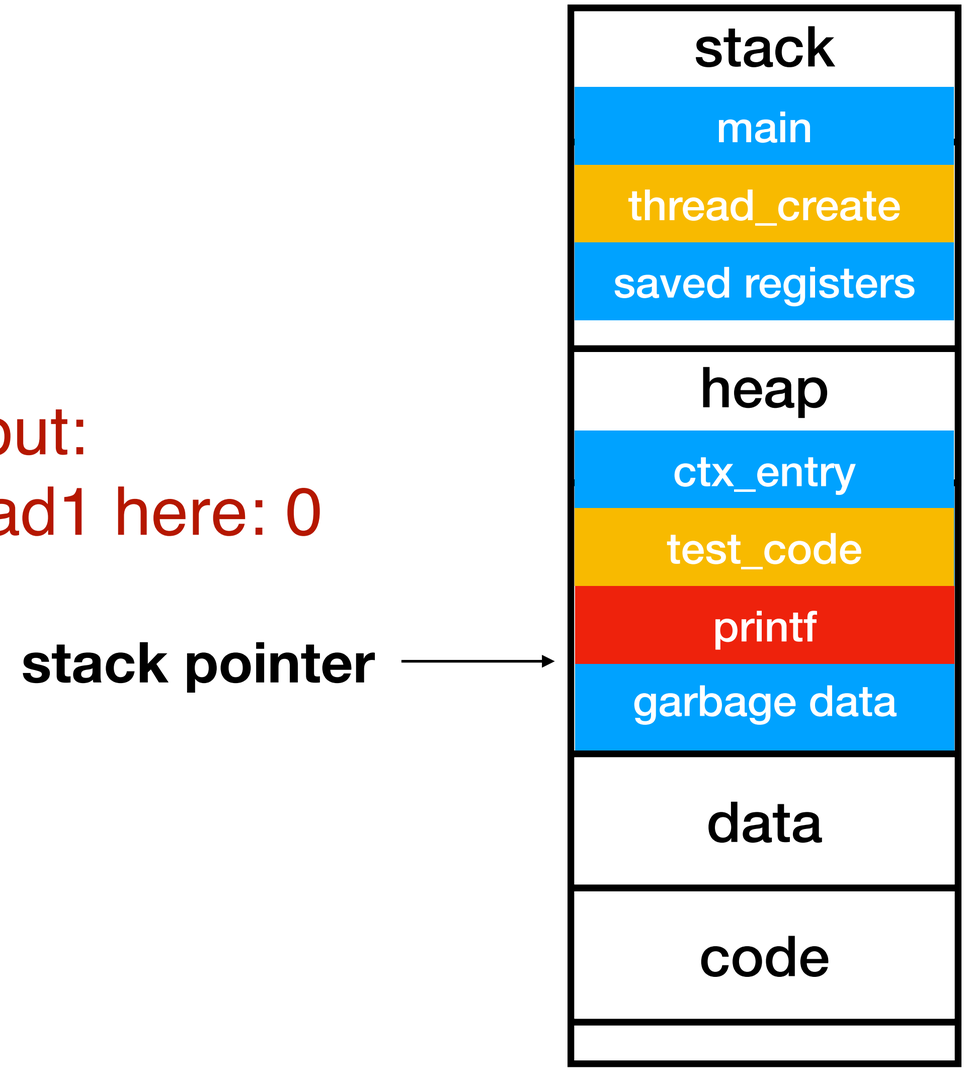
stack pointer →



Thread1 executes

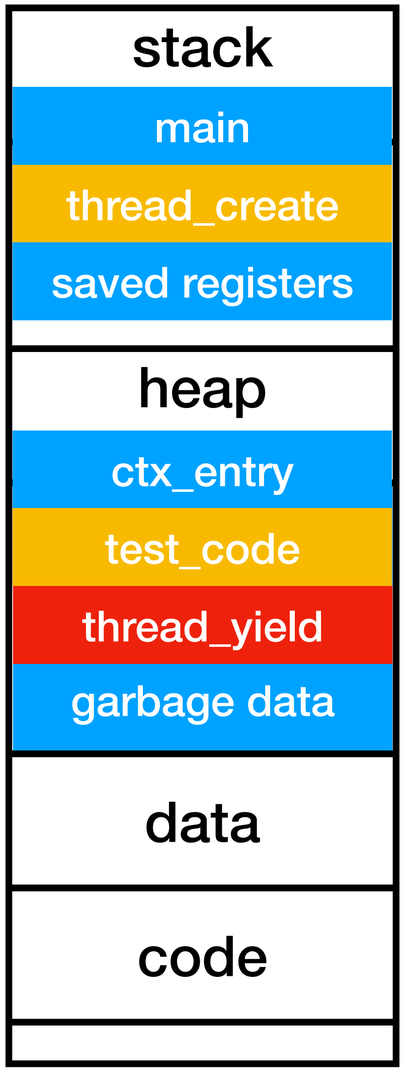
```
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
}

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

Output:
thread1 here: 0



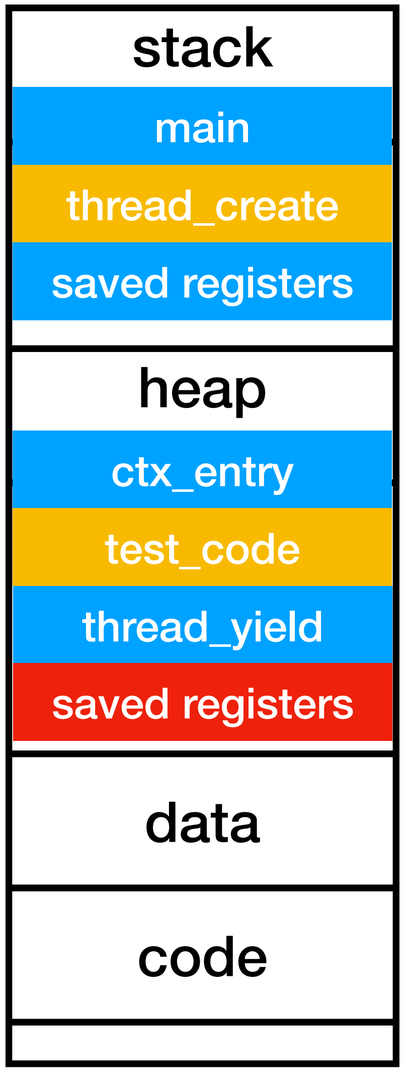
Thread1 yields

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();  // switch the context back to main thread
    }
    printf("%s done\n", arg);
    thread_exit();
}
```



Yield step 1/4

```
void test_code(void *arg) {
      …
      thread_yield();
      …
}

ctx_switch:  // step 1/4: thread_yield() calls ctx_switch()
    …        // save registers on the stack with store instructions
    mv sp, a1
    …
    ret
```



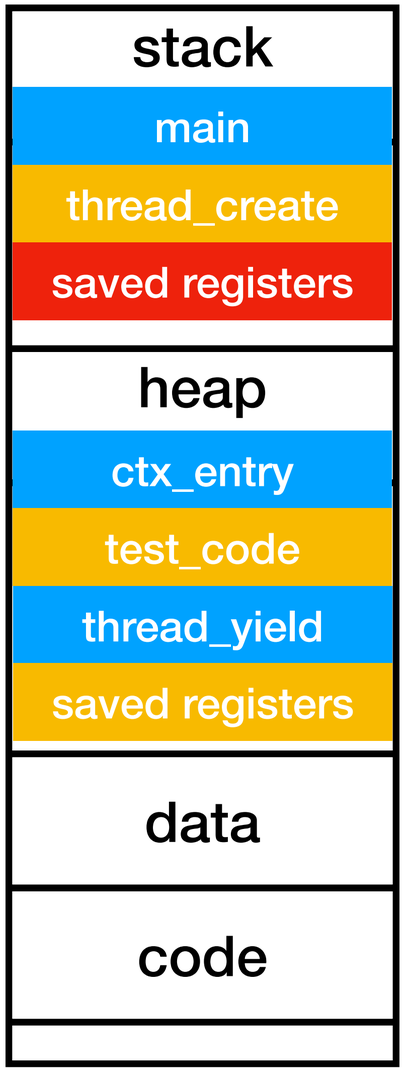
stack pointer →

Yield step 2/4

```
void thread_yield() {
    // read the stack pointer of the main thread
    // from the data structures maintained by
    // your code and pass it to ctx_switch()
}

ctx_switch:
    …
    mv sp, a1  // step 2/4: change the stack pointer
    …
    ret
```

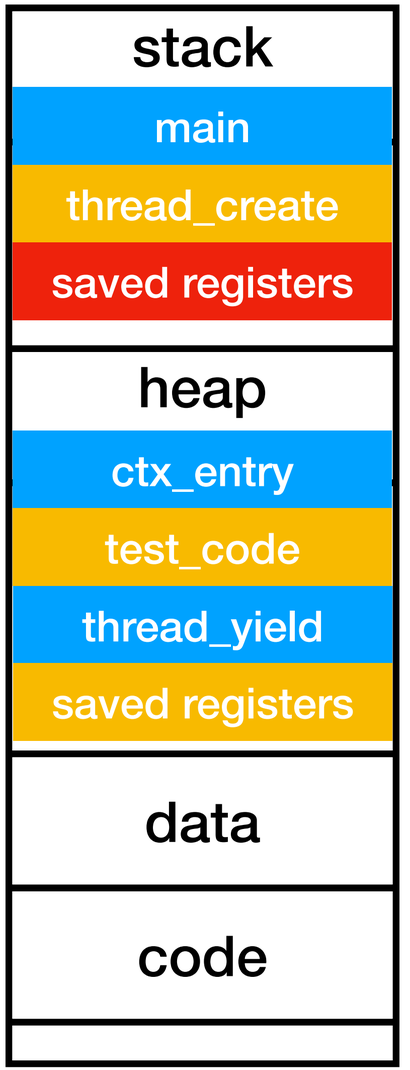
new stack pointer →

old stack pointer →



Yield step 3/4

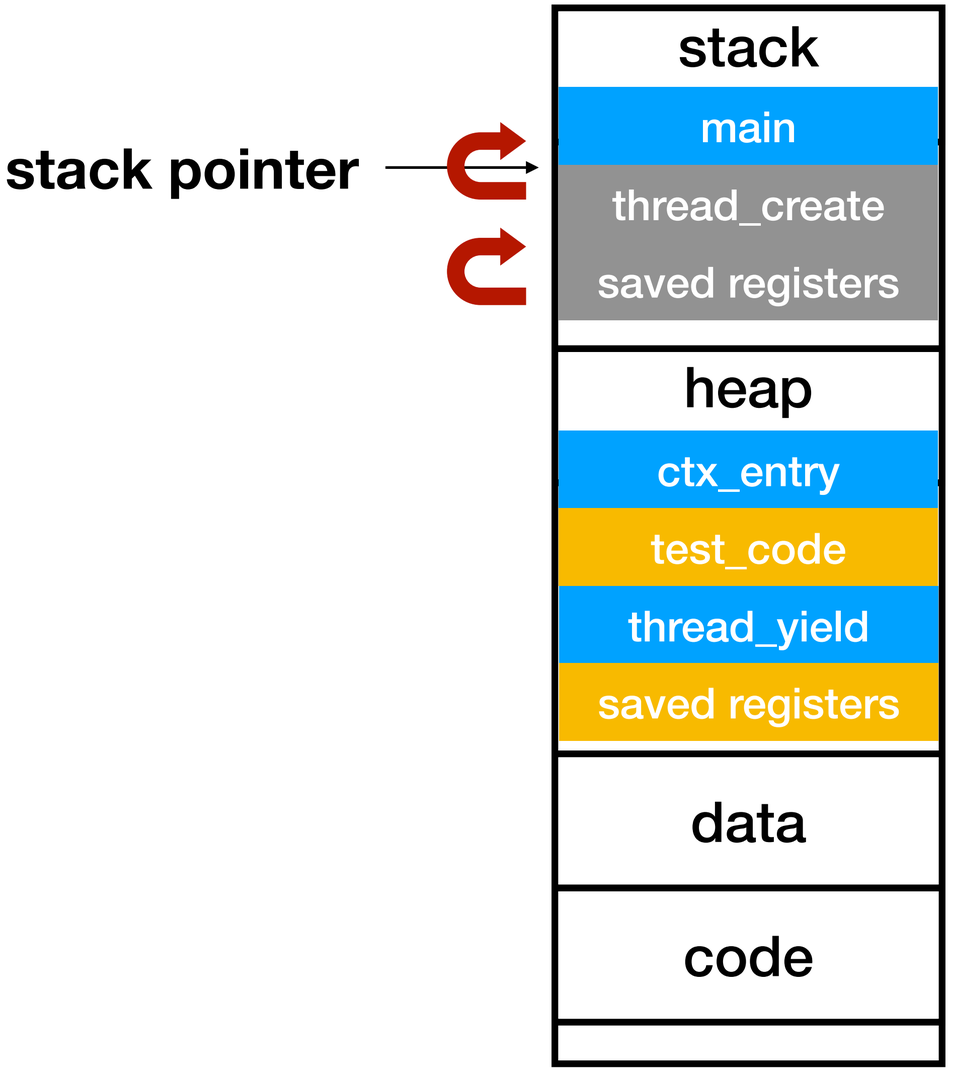
```
void test_code(void *arg) {
      …
      thread_yield();
      …
}

ctx_switch:
    …
    mv sp, a1
    …  // step 3/4: resume saved registers with load instructions
    ret
```



stack pointer

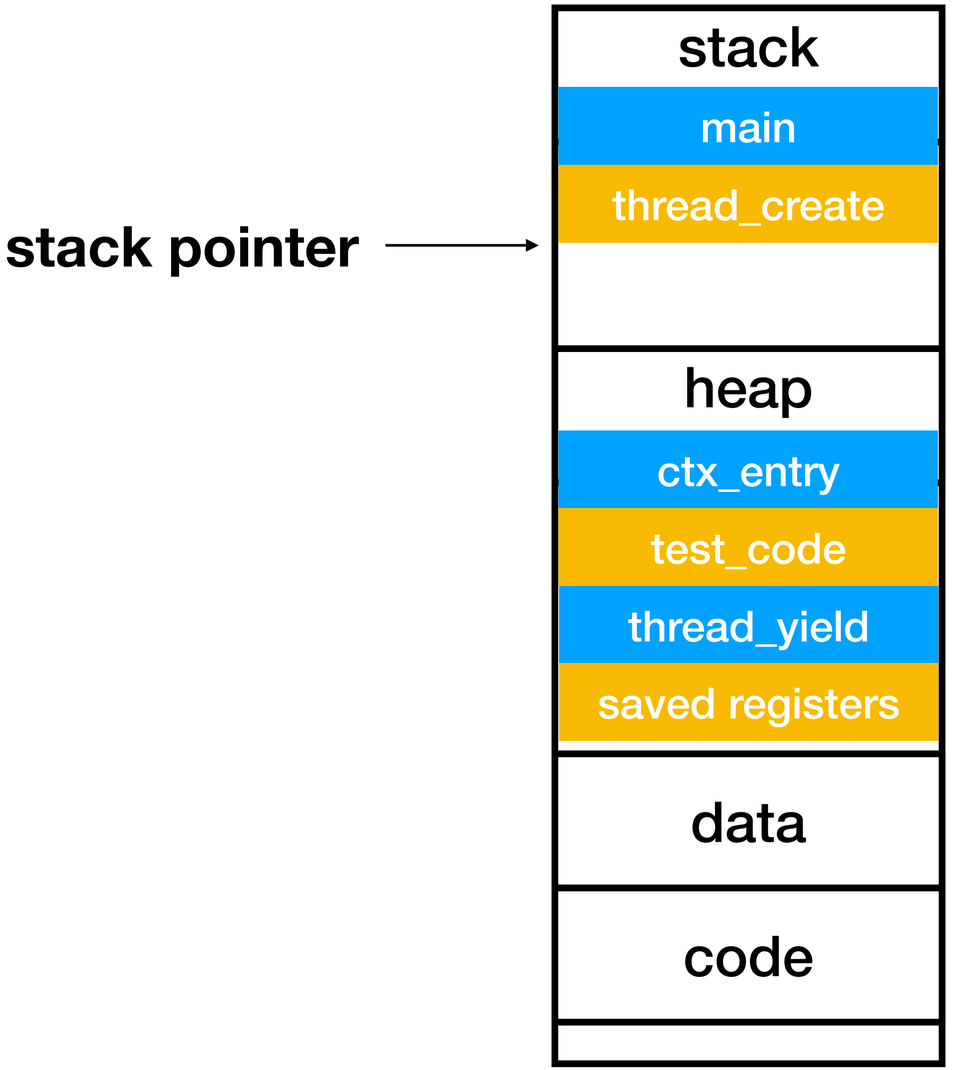
Yield step 4/4

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

ctx_switch:
    …
    mv sp, a1
    …
    ret  // step 4/4: return to thread_create()
thread_create:
    …
    ret  // step 4/4: return to main()
```



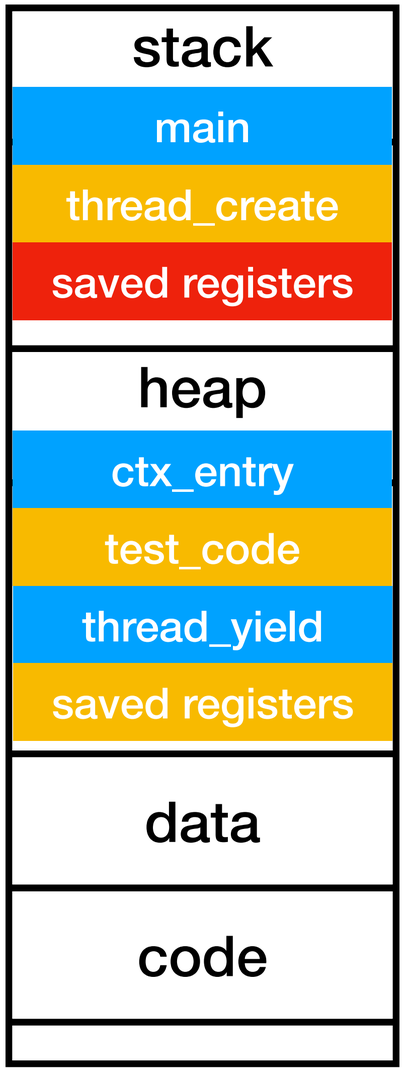
Main thread continues

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

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```



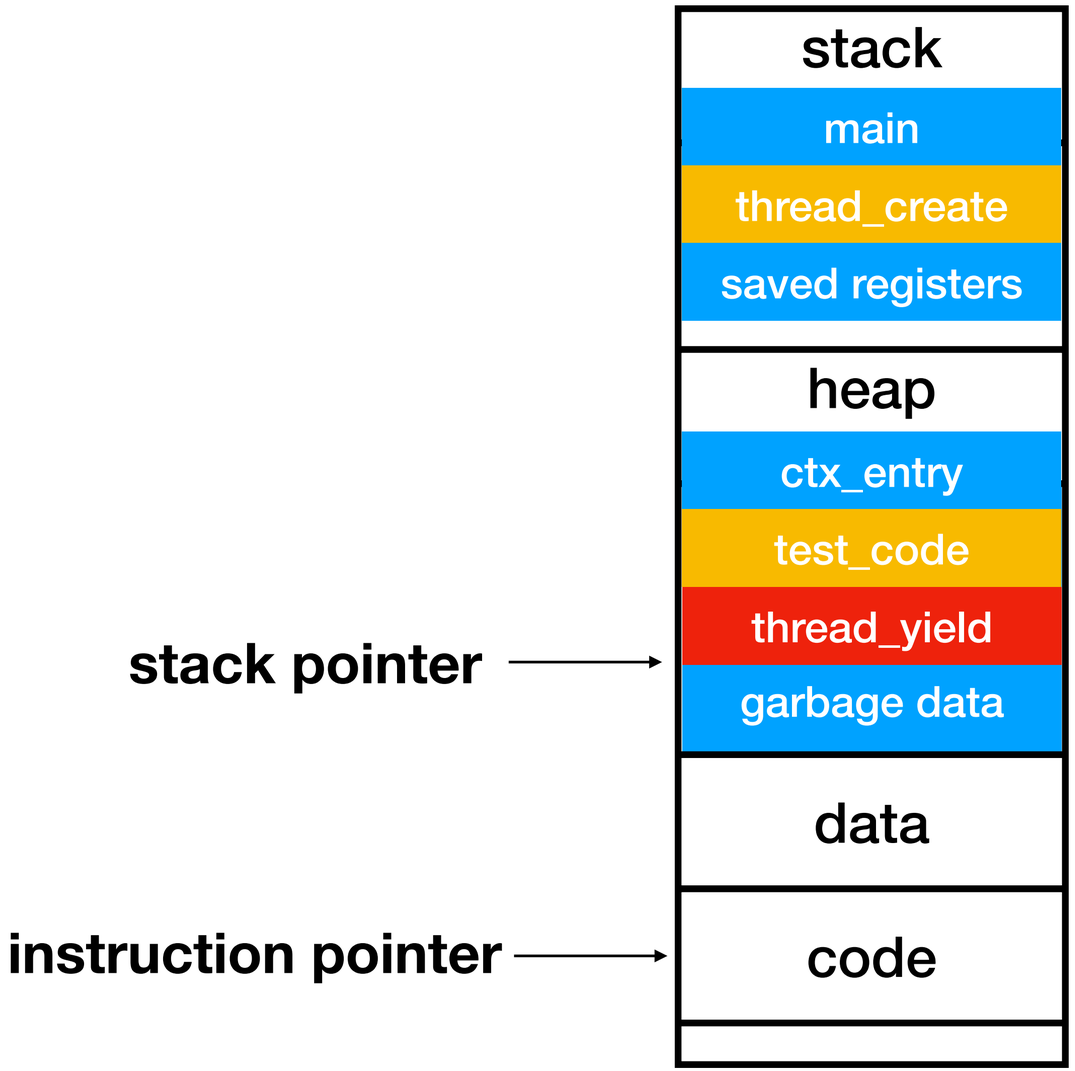
Two threads

- The two user-level threads share
  - memory address space
  - code section (i.e., instruction pointer)
    - `thread_create()`, `thread_yield()`, ...
- They have different stack and stack pointers.
- Consider, in zoom, one thread controls the microphone and another controls the camera.

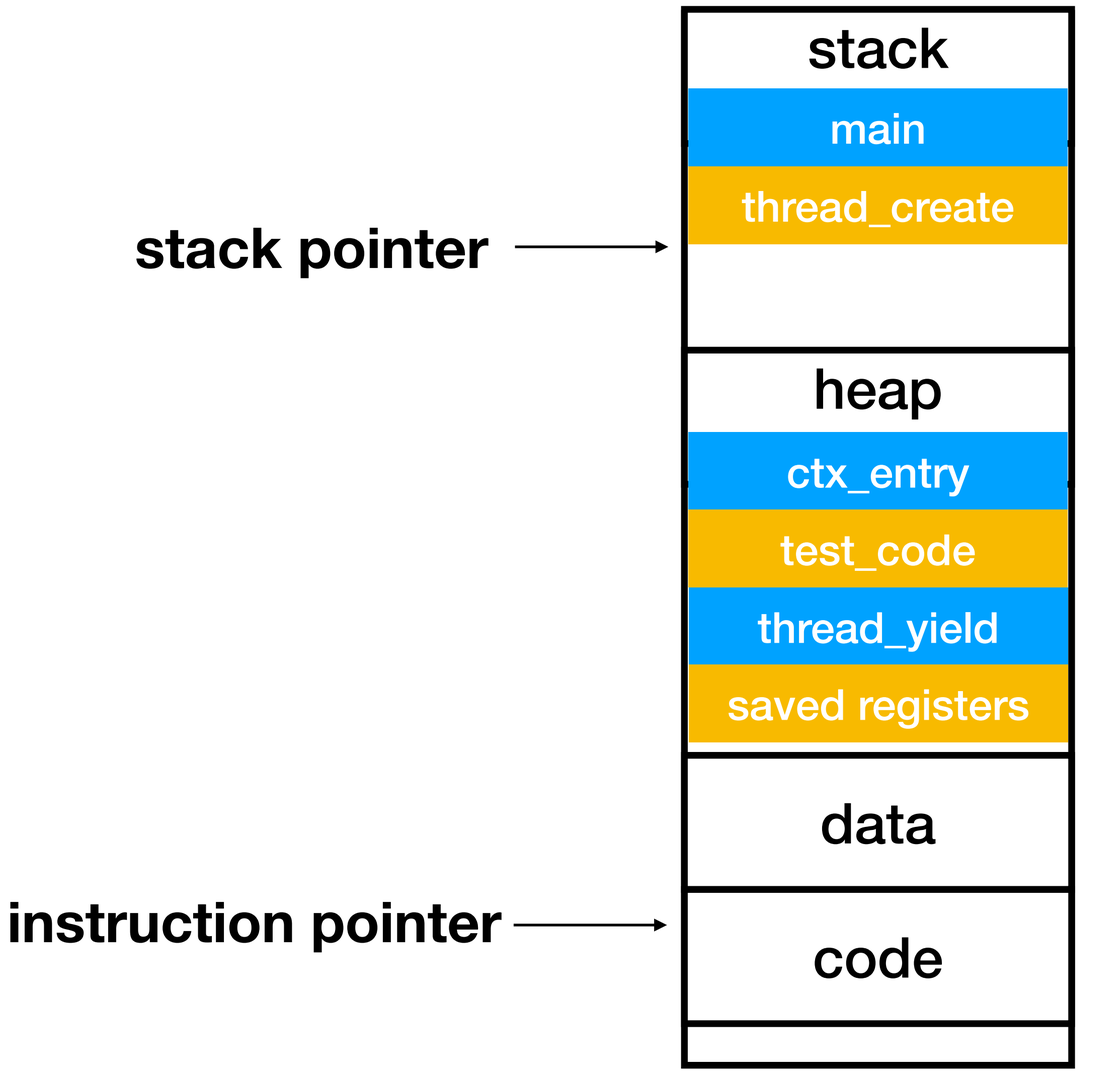


User-level thread vs. Kernel thread

- P1 (user-level): without CPU support
  - Thread 1 needs to call `thread_yield()`. (Non-preemptive)
- P2 (kernel): with CPU support
  - A CPU register holds the address of `thread_yield()`.
  - CPU calls `thread_yield() when interrupted`. (Preemptive)

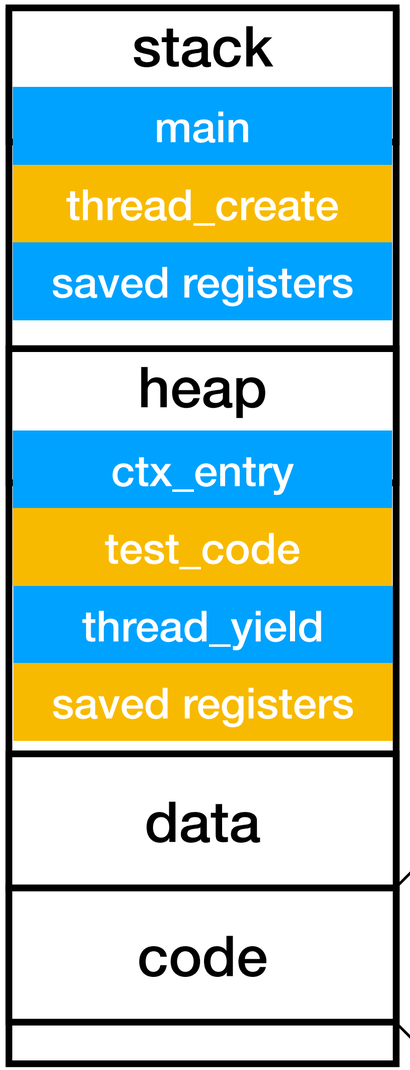


CPU in the **context** of **thread1**



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

Precisely, when does context switch happen?



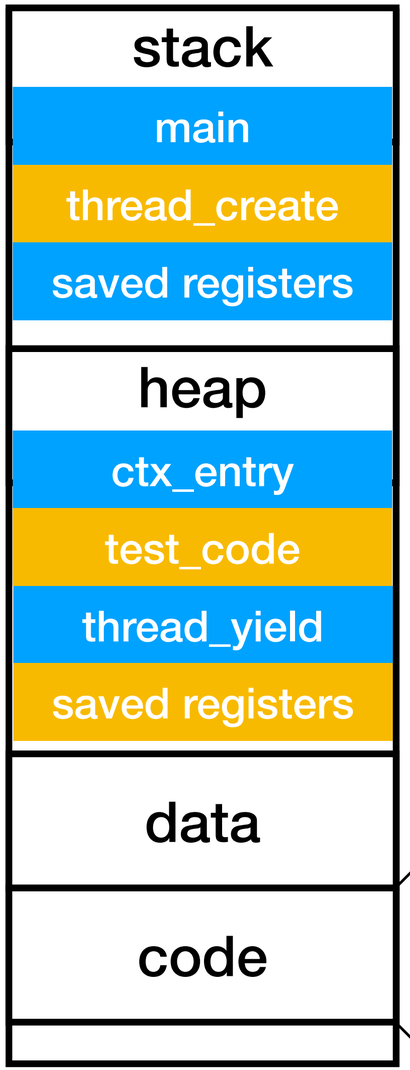
stack pointer →

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

```
ctx_switch:
    …
    mv sp, a1
    …
    ret
```

instruction pointer

Precisely, when does context switch happen?



stack pointer →

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

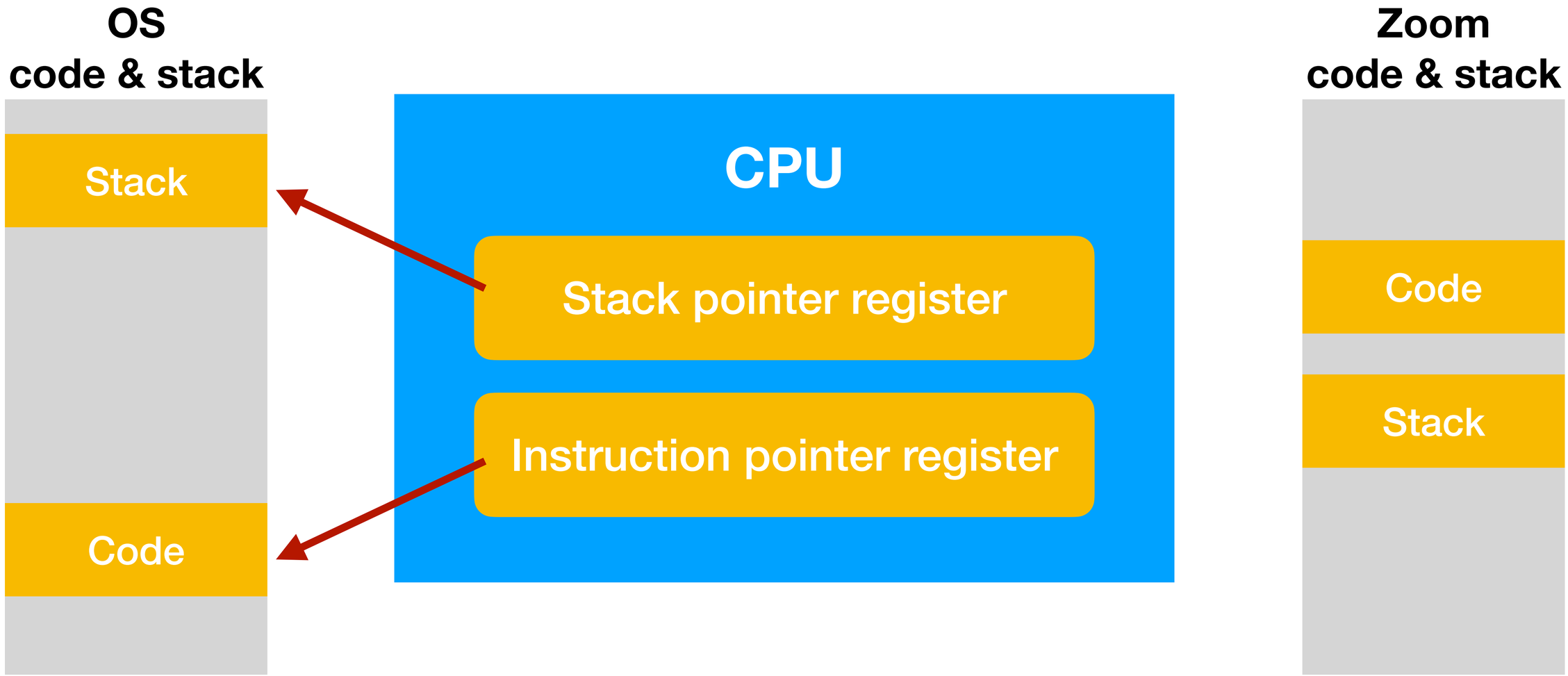
```
ctx_switch:
    …
    mv sp, a1
    …
    ret
```

← **instruction pointer**

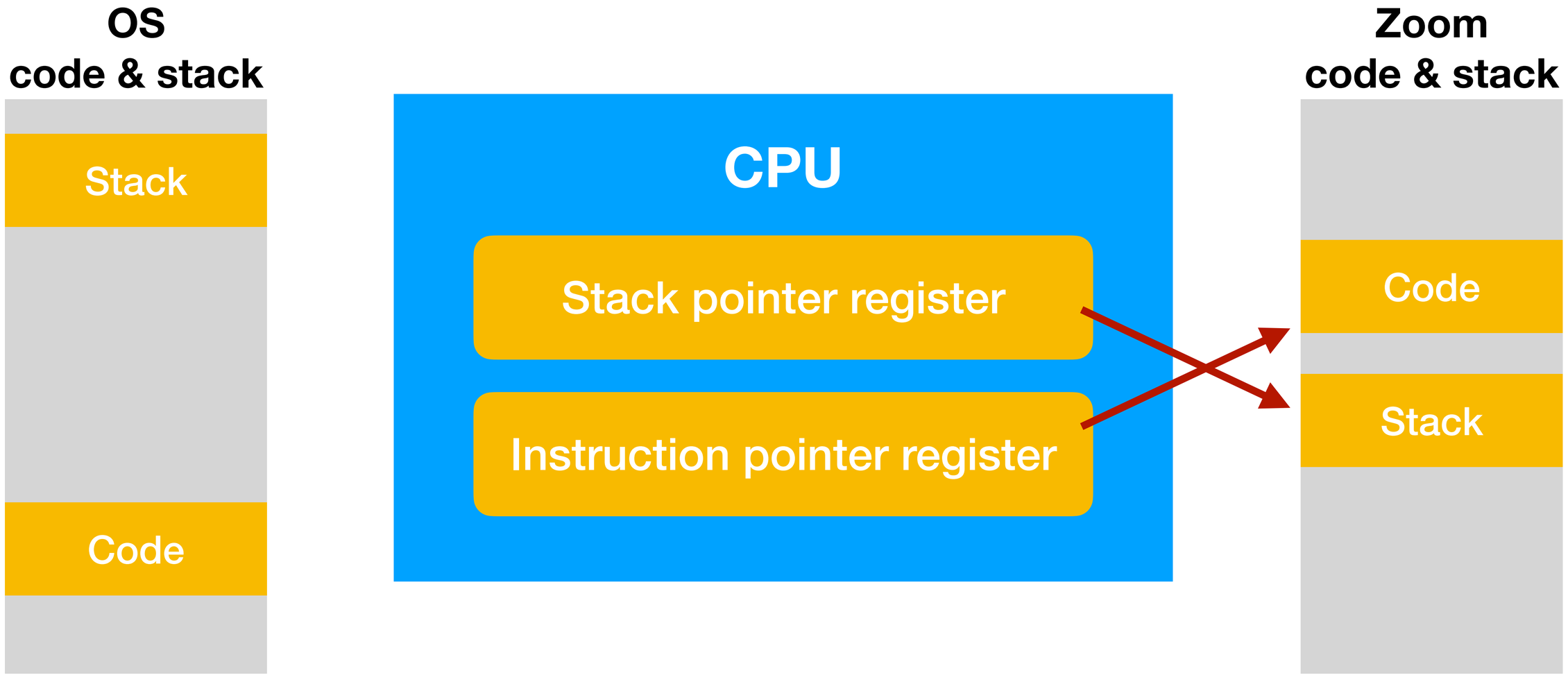
Recap

context = memory abstraction
+ stack pointer + instruction pointer

CPU in the context of OS



CPU in the context of Zoom



context = memory abstraction
+ stack pointer + instruction pointer

Question

Why didn’t the definition mention registers?

Registers

General-purpose registers

a0, a1, …
t0, t1, …
…

Special-purpose registers

stack pointer register
instruction pointer register
…

Memory hierarchy

General-purpose registers

L1-L3 CPU cache

DDR main memory

SSD or hard disk

Implementation vs. Abstraction

General-purpose registers

L1-L3 CPU cache

DDR main memory

SSD or hard disk

These implementations provide

a memory abstraction:

Content	1st byte	2nd byte	3rd byte	……	2^32th byte
Address	0x0000 0000	0x0000 0001	0x0000 0002	……	0xFFFF FFFF

Take-away #1

Distinguish abstractions from implementations.

Abstractions should persist while implementations continuously change.

Context: from implementation to abstraction

General-purpose registers

L1-L3 CPU cache

DDR main memory

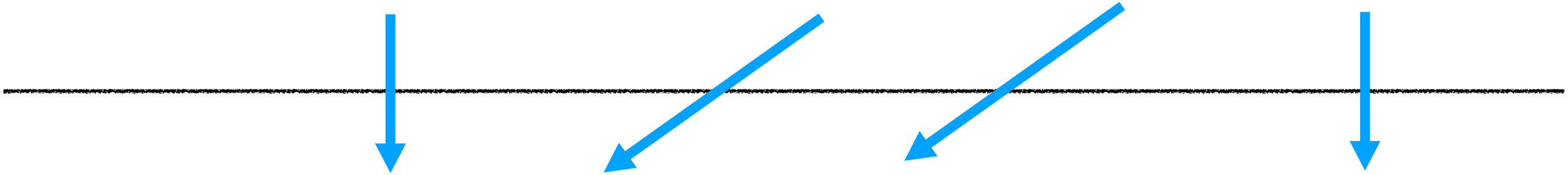
SSD or hard disk

Special-purpose registers

Page table base register

Stack pointer register

Instruction pointer register

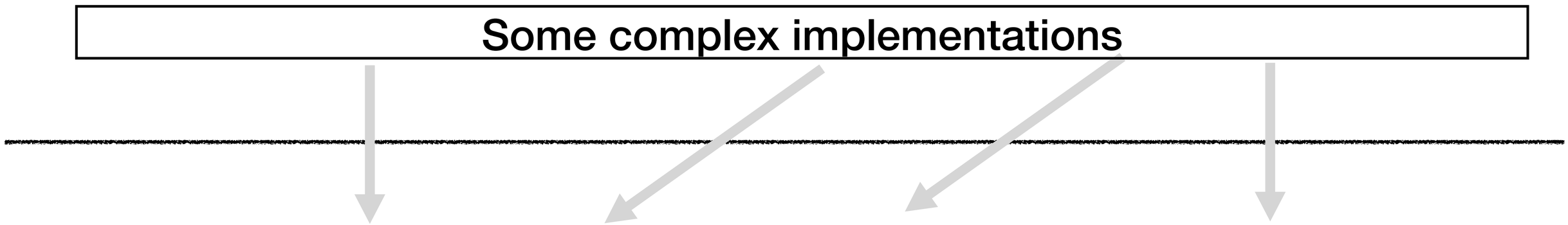


memory abstraction

stack pointer

instruction pointer

Abstractions are simple



A table with 2 rows + 2 special positions in the table.

Content	1st byte	2nd byte	3rd byte	……	2^32th byte
Address	0x0000 0000	0x0000 0001	0x0000 0002	……	0xFFFF FFFF

Take-away #2

Abstractions make definitions simple.

Good systems design should aim for simplicity.

Summary

- An example of multi-threading in P1:
  - how to create a thread; how to switch between threads
- User-level vs. Kernel threading:
  - non-preemptive vs. preemptive
  - users call yield vs. CPU calls yield during an interrupt
- Distinguish implementations from abstractions
  - low-level: memory hierarchy, special-purpose registers
  - high-level: memory abstraction, stack pointer, instruction pointer
- Using abstractions makes definitions simple.

Homework

- P1 has been released on CMSx and is due on Sep 28.
- P1 has three parts
  - multi-threading (today)
  - synchronization + testing suite (next week)

Next lecture

- More details of the multi-threading API
  - init, create, yield, exit
- Semaphores for synchronization
- Producer / consumer using semaphores
- Introduction to the EGOS teaching operating system