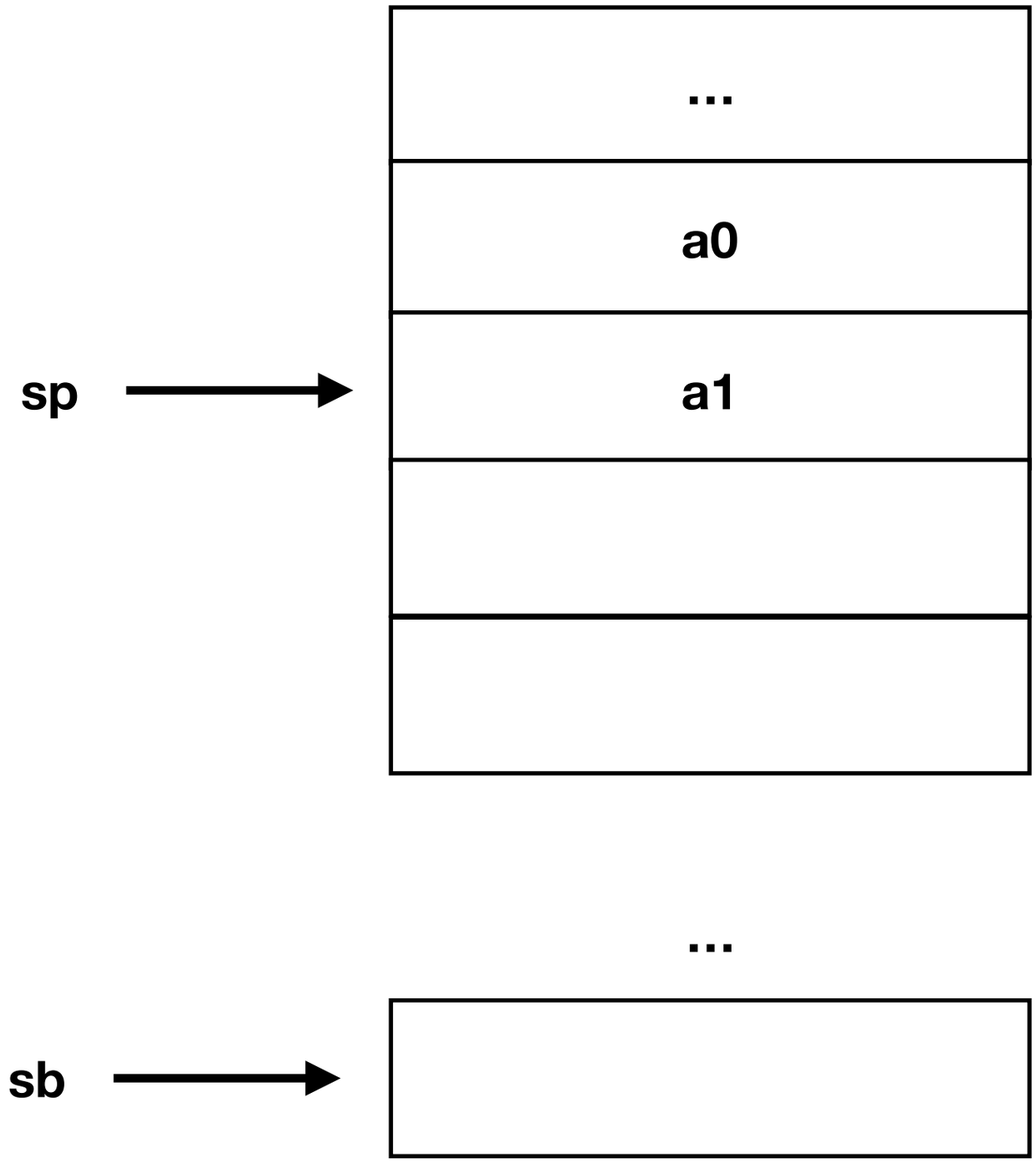
P1 - Non-Preemptive Thread Scheduler

CS 4411
Feb 2. 2018
Drew Zagieboylo

The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer

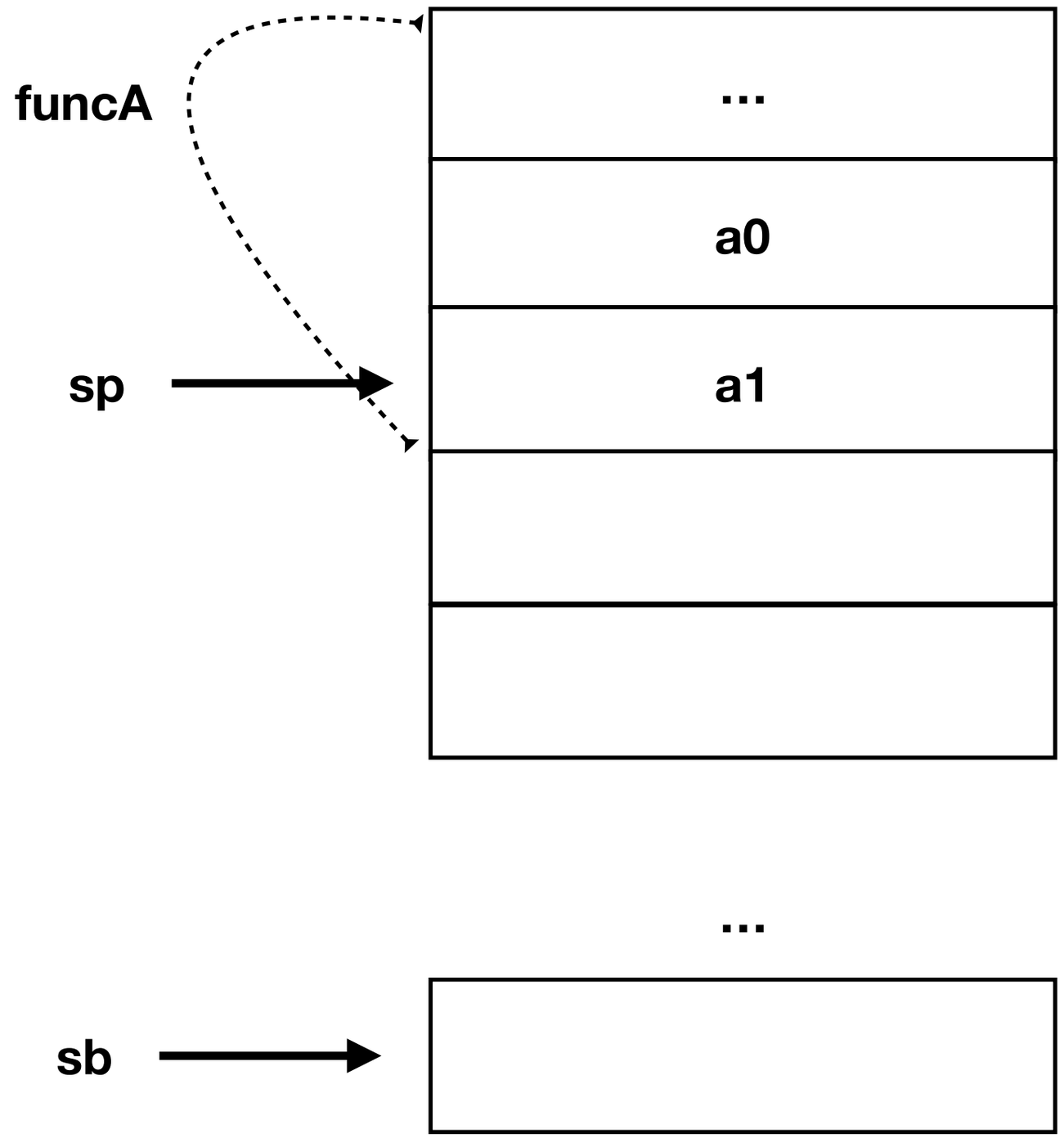


The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer

```
//funcA

x = 3;
y = 2;
z = add(x, y);
return z;
```

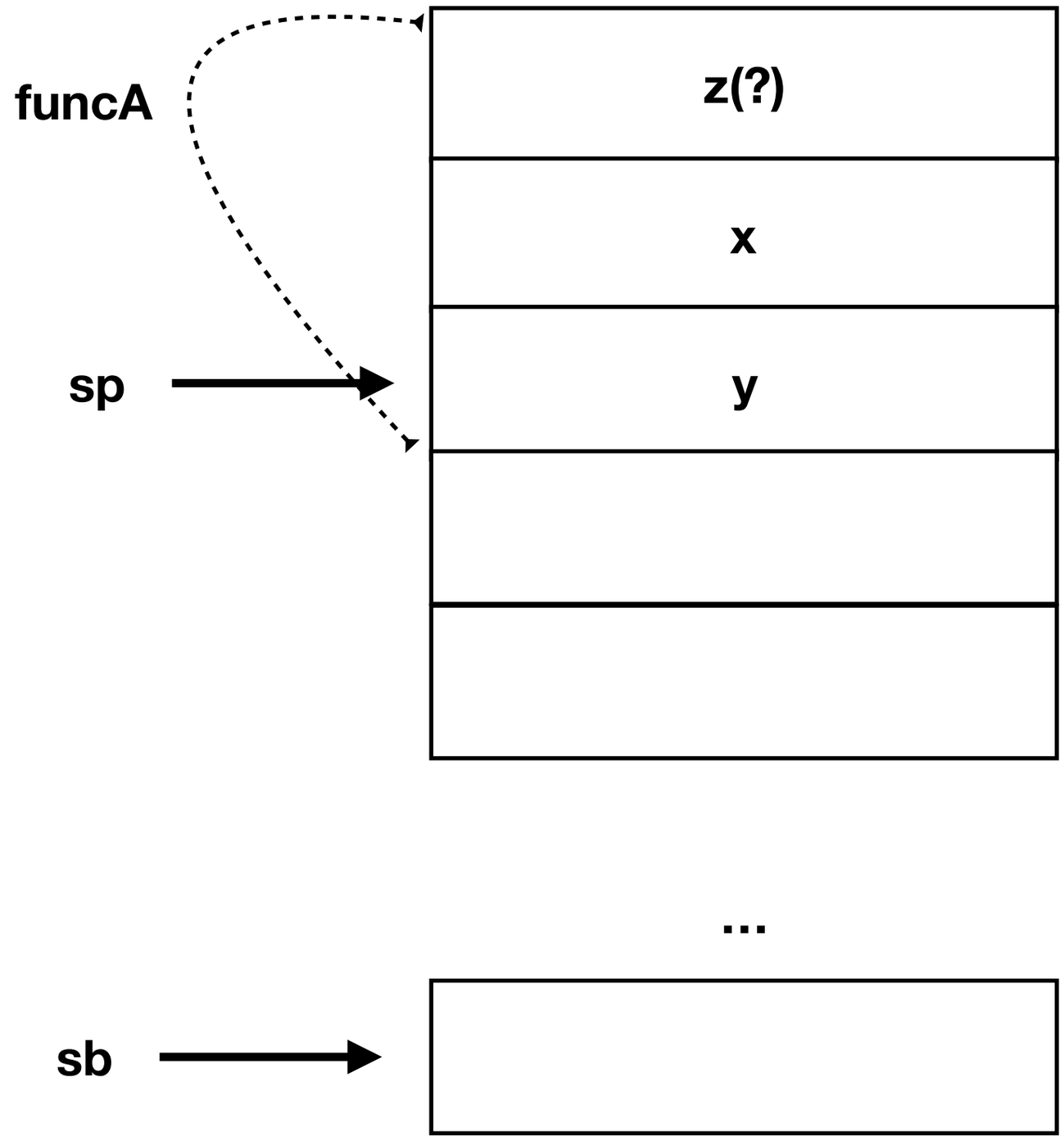


The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer

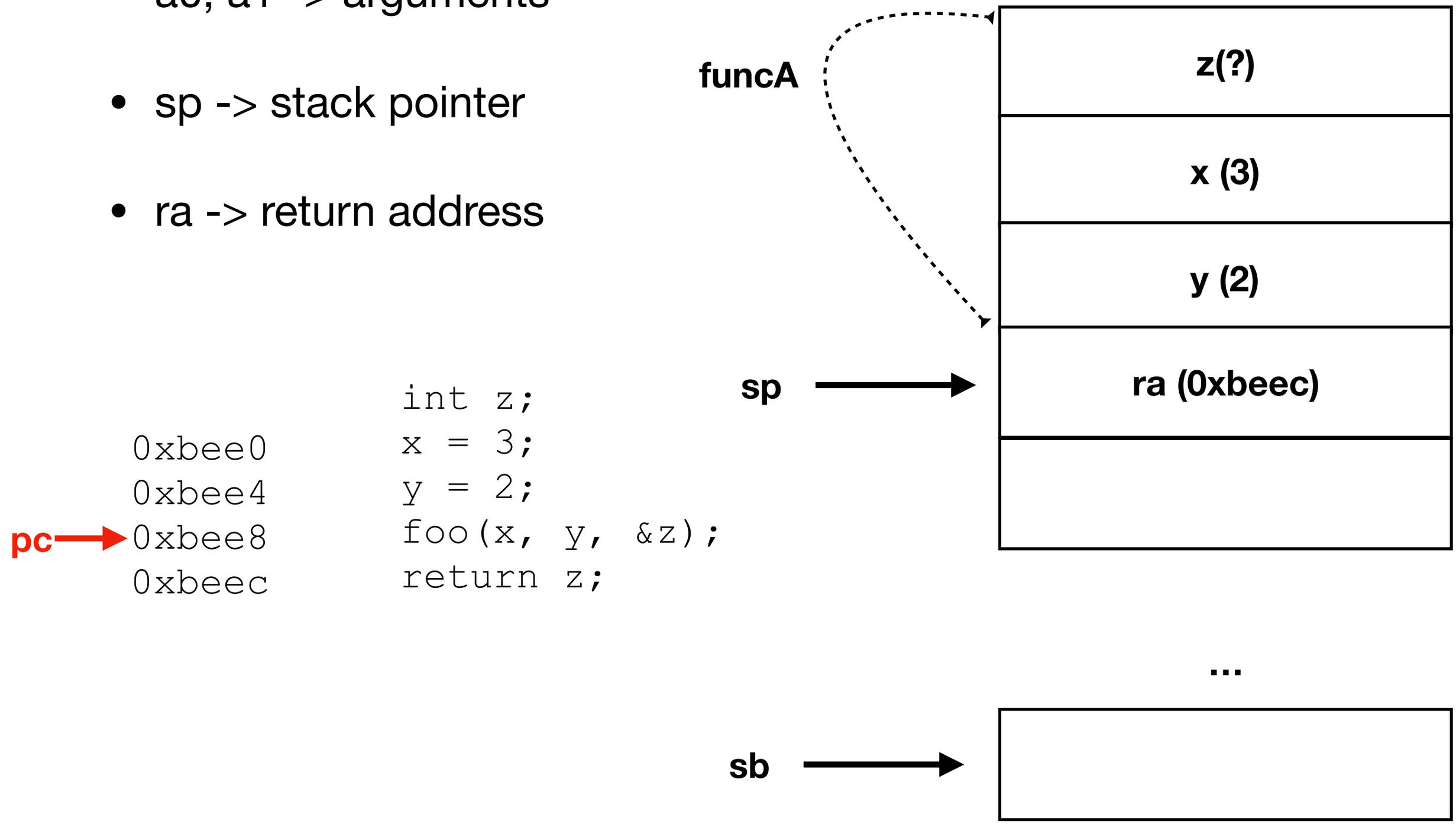
```
//funcA

int z;
x = 3;
y = 2;
foo(x, y, &z);
return z;
```



The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer
- ra -> return address

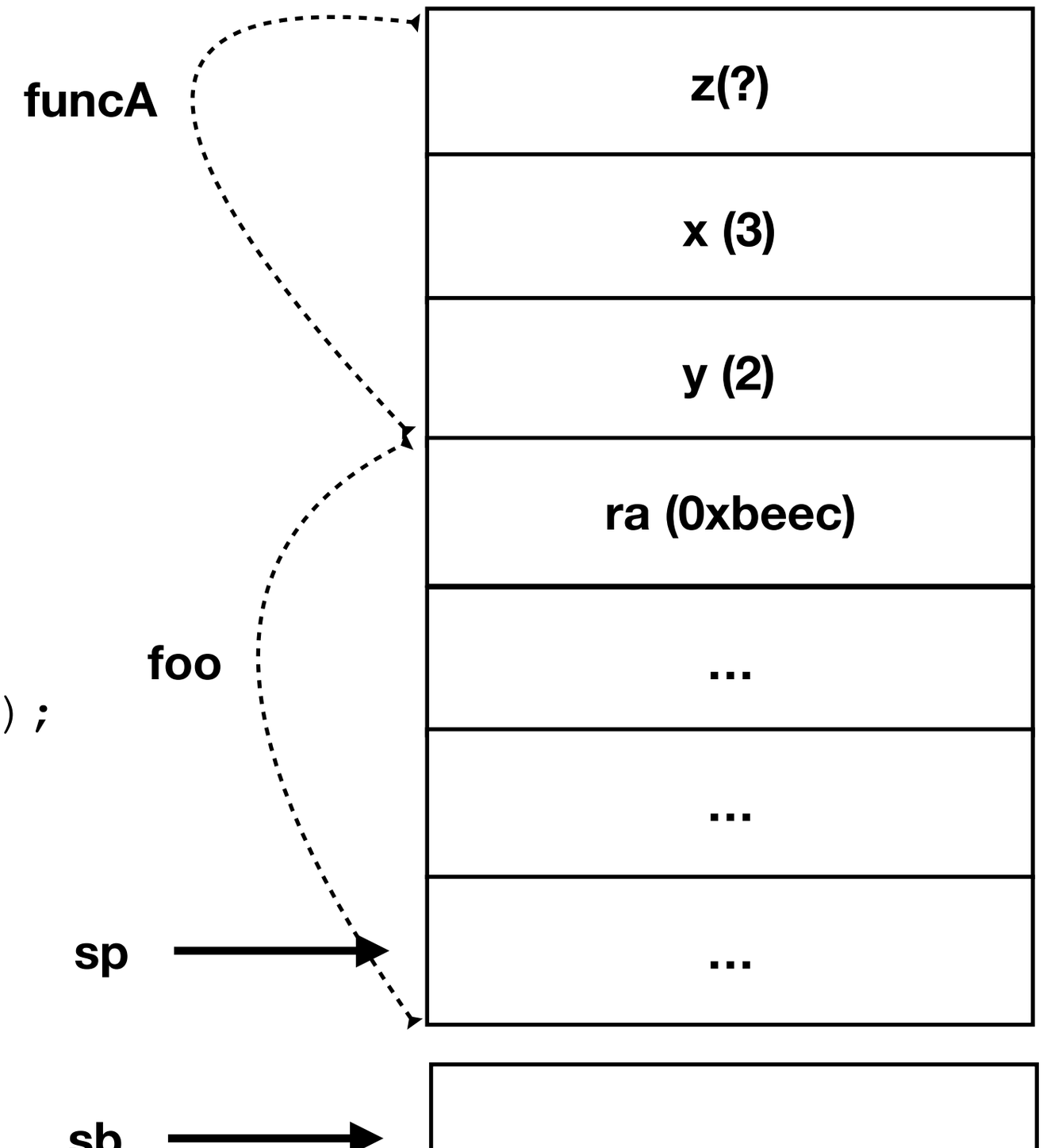


The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer
- ra -> return address

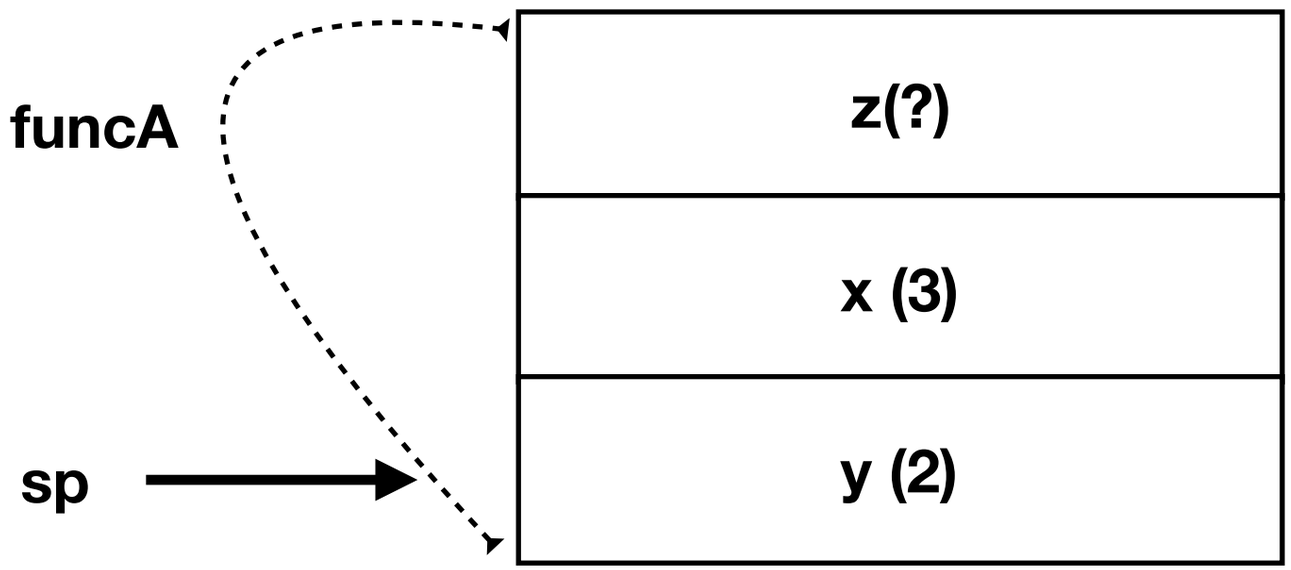
```
                int z;
0xbee0          x = 3;
0xbee4          y = 2;
0xbee8          foo(x, y, &z);
0xbeec          return z;
```

pc → &foo

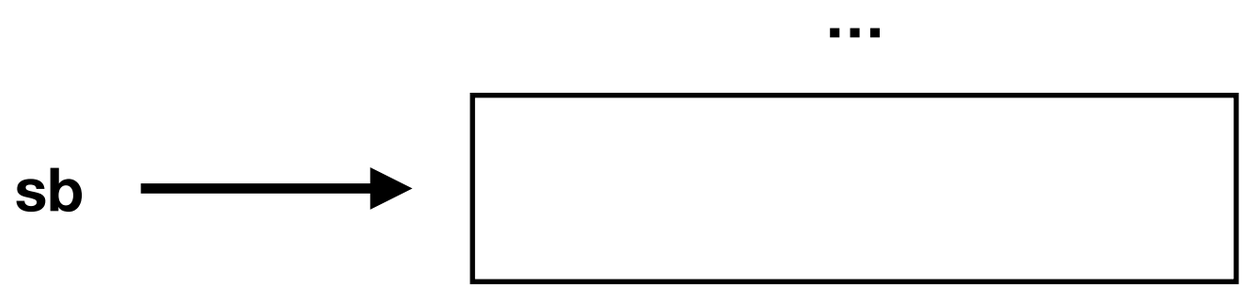


The Stack - Review

- a0, a1 -> arguments
- sp -> stack pointer
- ra -> return address




```
                int z;
    0xbee0      x = 3;
    0xbee4      y = 2;
    0xbee8      foo(x, y, &z);
pc  0xbeec      return z;
```



The Stack - Review

- Every Process/Thread has its own *Stack*
- Every Function has its own *Stack Frame*
- The stack grows *down* as functions are called; returns go back *up* the stack
- *Returning* from a function is a *jump* to the *return address* on the stack

Minithreads

- minithreads are your version of *threads*
- Allows timesharing execution of a single CPU
- Independent stacks, pcs, register values, etc.
- Will need a *Thread Control Block* to manage thread info.

Minithread API

- minithread_fork(process, argument)
    - create a new thread and run *process*
- minithread_yield()
    - allow a different minithread to execute
- minithread_stop()
    - de-schedule this minithread (stop running until someone explicitly starts you again)
- (a few others that are similar to the above)

Context Switching

- Need to save all current minithread state (registers, pc, sp, etc.)
- Load new minithread's state from where it was saved, and start executing.

- Where to save state?
  - The stack! (mostly)
- We have provided primitives for context switching:
  - machineprimitives.h/c
  - machineprimitives_x86_86.c/S

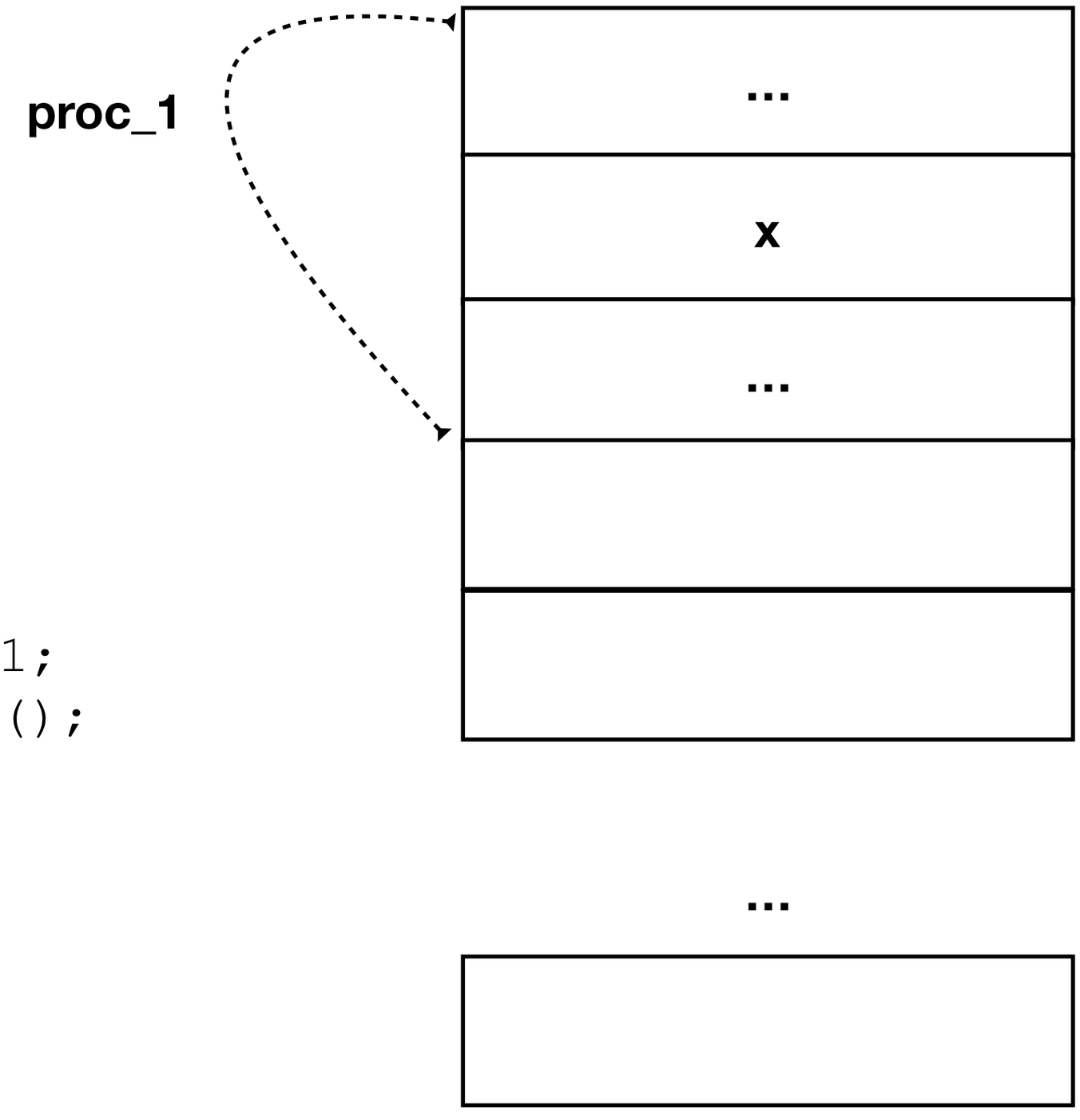
mt_switch

- minithread_switch(old_sp*, new_sp*)
- saves current processor *sp* to old_sp
- loads value in new_sp to processor *sp*
- reloads registers and pc from stack

Context Switching

```
//proc_1

      0xbee0      while (1){
      0xbee4          x = x + 1;
pc →  0xbee8          mt_yield();
      0xbeec      }
```

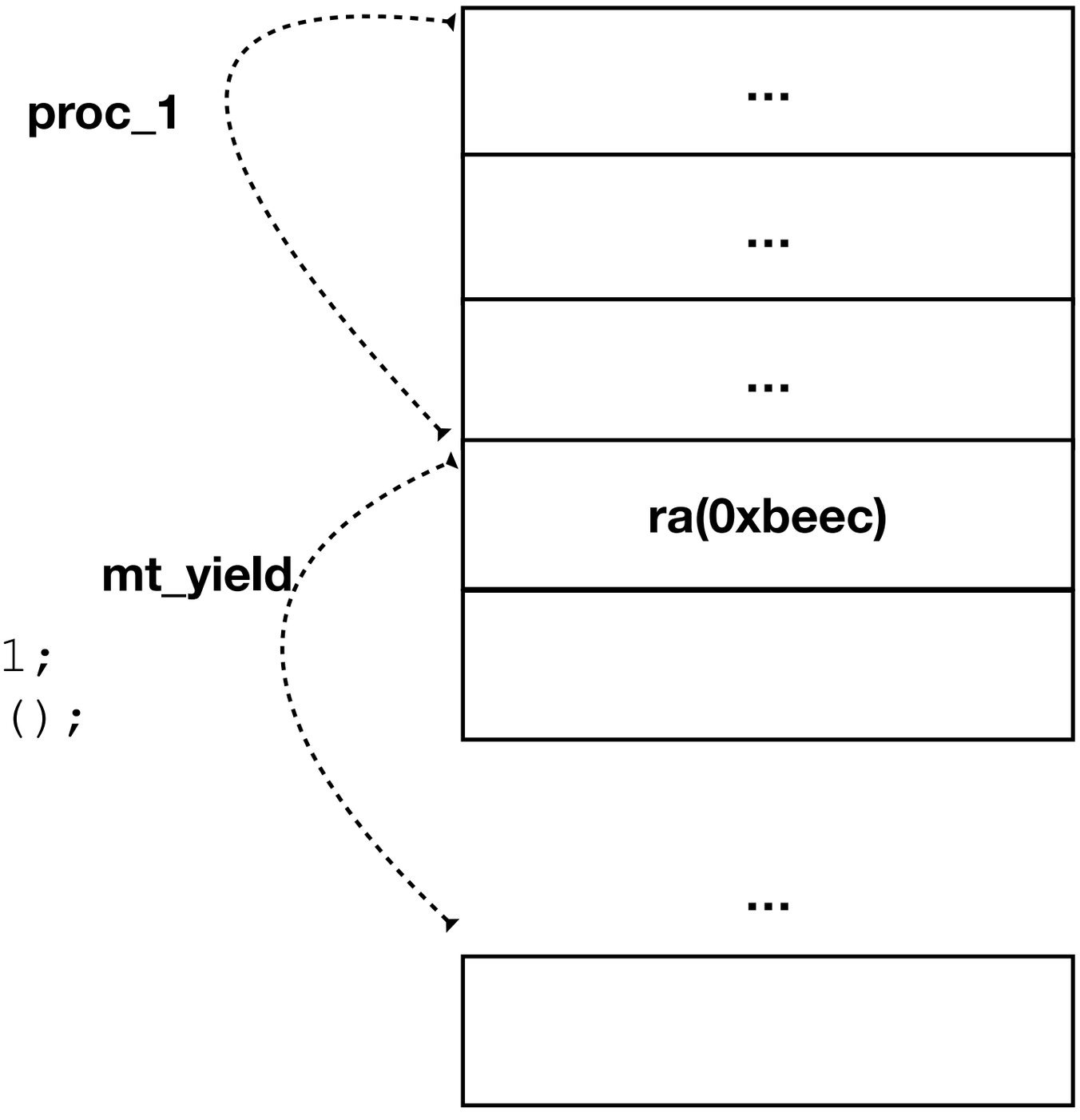


Context Switching

```
//proc_1

    0xbee0      while (1){
    0xbee4          x = x + 1;
    0xbee8          mt_yield();
    0xbeec      }
```

pc → &mt_yield

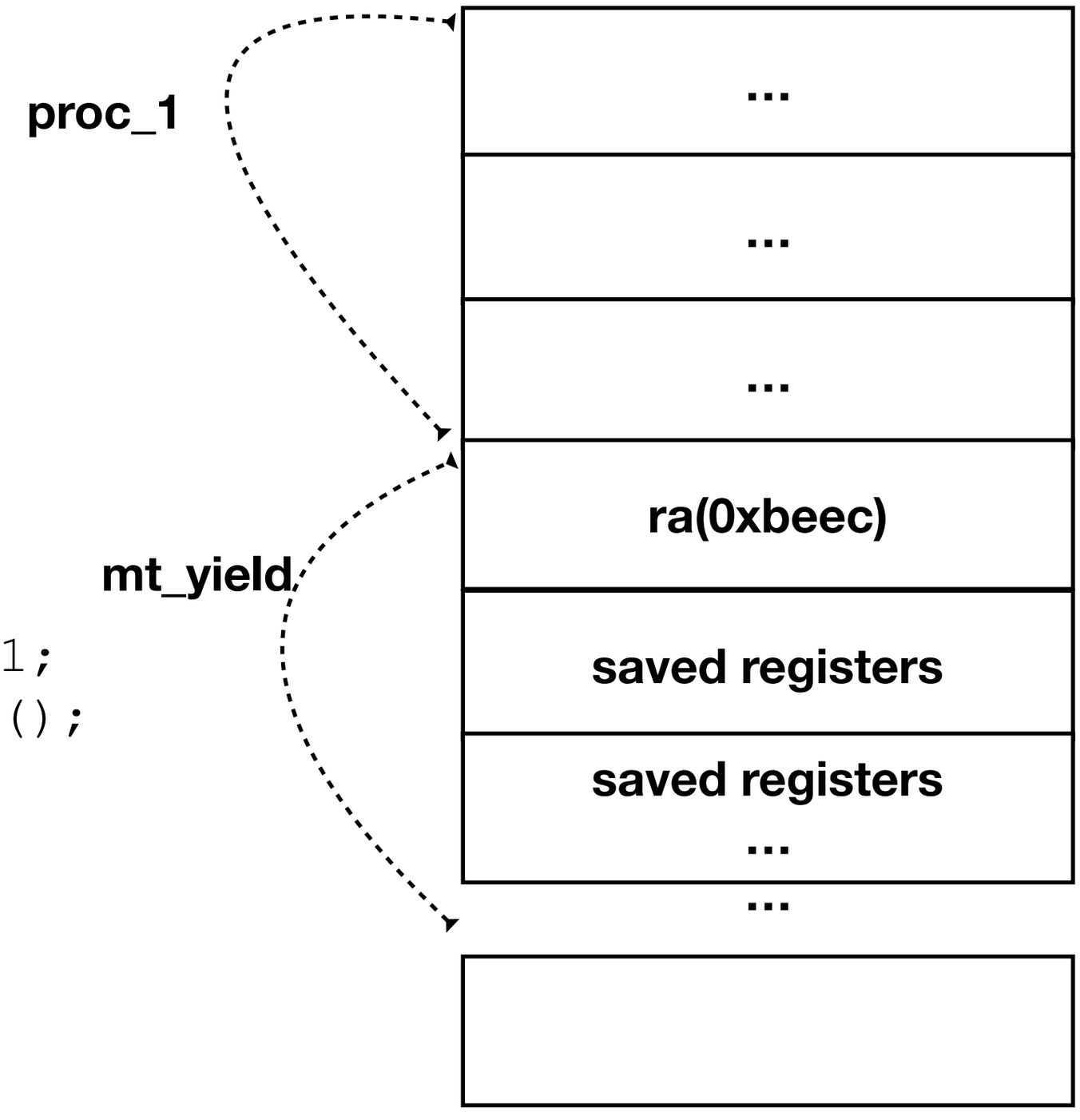


Context Switching

```
//proc_1

0xbee0      while (1){
0xbee4          x = x + 1;
0xbee8          mt_yield();
0xbeec      }
```

pc → &mt_yield



Context Switching

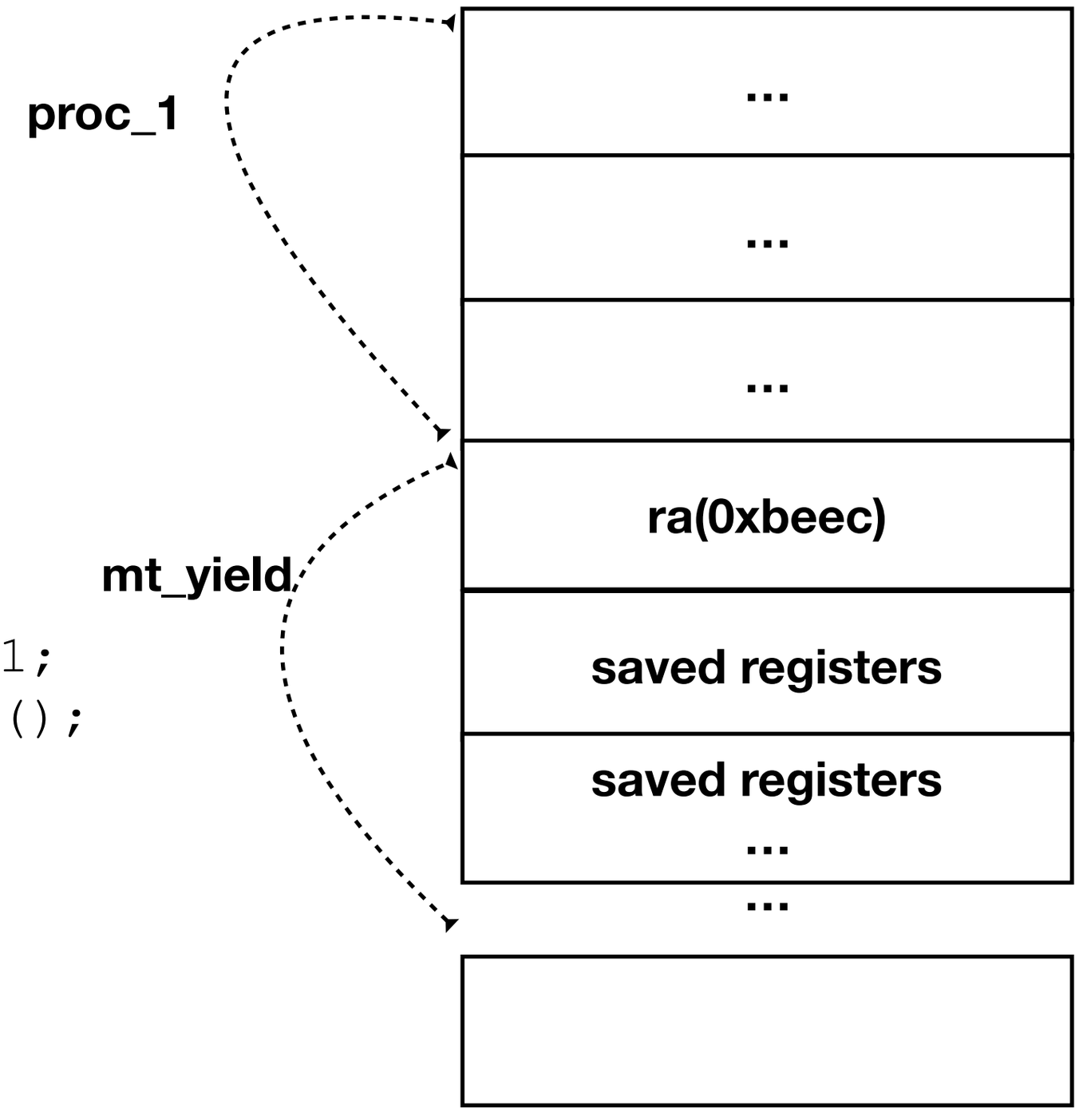
Meanwhile,
somewhere else in memory..

tcb_proc1

```
//proc_1

0xbee0      while (1){
0xbee4          x = x + 1;
0xbee8          mt_yield();
0xbeec      }
```

pc → **&mt_yield**



Context Switching

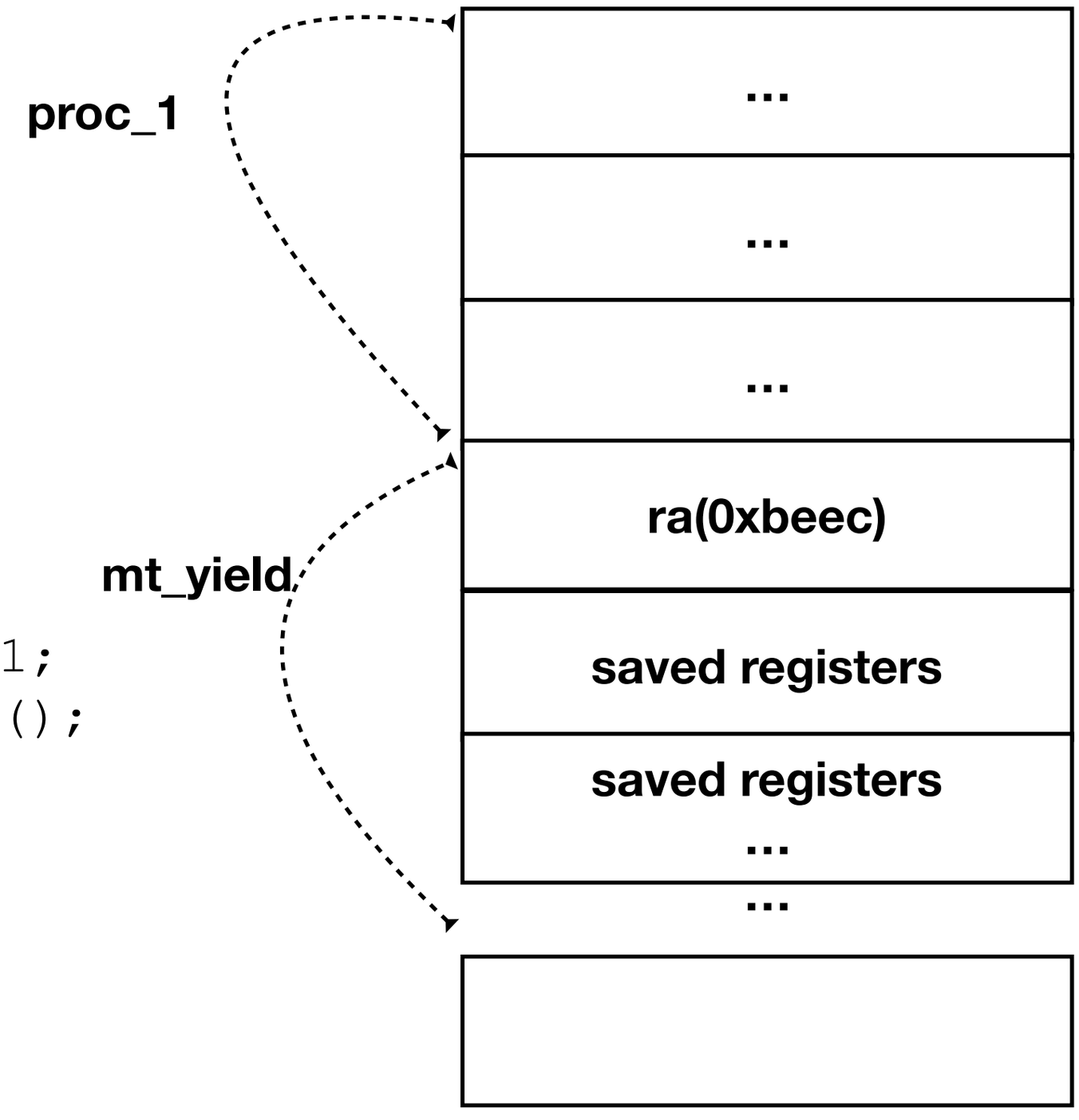
Meanwhile,
somewhere else in memory..

tcb_proc1->save sp

```
//proc_1

0xbee0      while (1){
0xbee4          x = x + 1;
0xbee8          mt_yield();
0xbeec      }
```

pc → **&mt_yield**



Context Switching

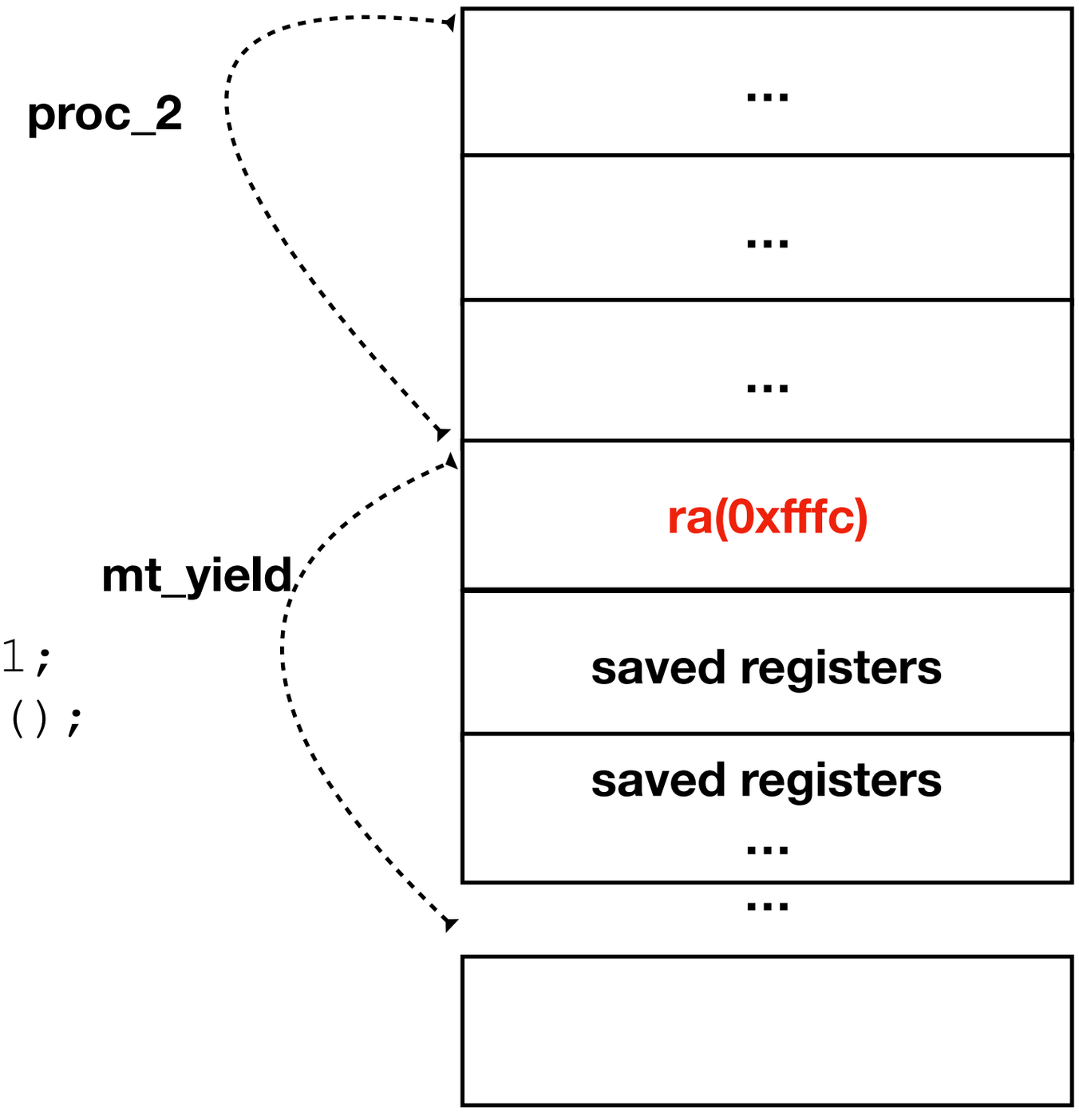
Meanwhile,
somewhere else else in memory..

tcb_proc2->load sp

```
//proc_2

0xfff0      while (1){
0xfff4          x = x - 1;
0xfff8          mt_yield();
0xfffc      }
```

pc → **&mt_yield**



Context Switching

Meanwhile,
somewhere else else in memory..

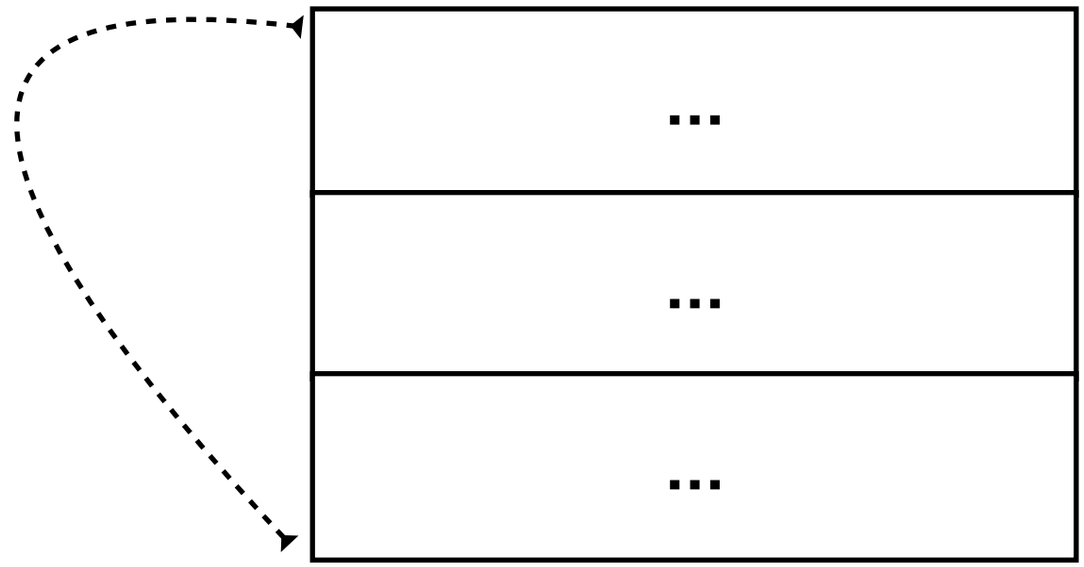
tcb_proc2->load sp

```
//proc_2
```

```
0xfff0      while (1) {
0xfff4          x = x - 1;
0xfff8          mt_yield();
0xfffc      }
```

pc →

proc_2



...

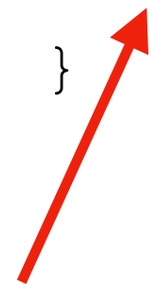
Context Switching

```
//proc_1

pc = 0xbee0      while (1){
pc = 0xbee4          x = x + 1;
pc = 0xbee8          mt_yield();
pc = 0xbeec      }
```

```
//proc_2

pc = 0xfff0      while (1) {
pc = 0xfff4          x = x - 1;
pc = 0xfff8          mt_yield();
pc = 0xfffc      }
```



Context Switching

- Where do non-running minithreads (tcbs) go?
- A Queue!
- For p1 -> round robin scheduling

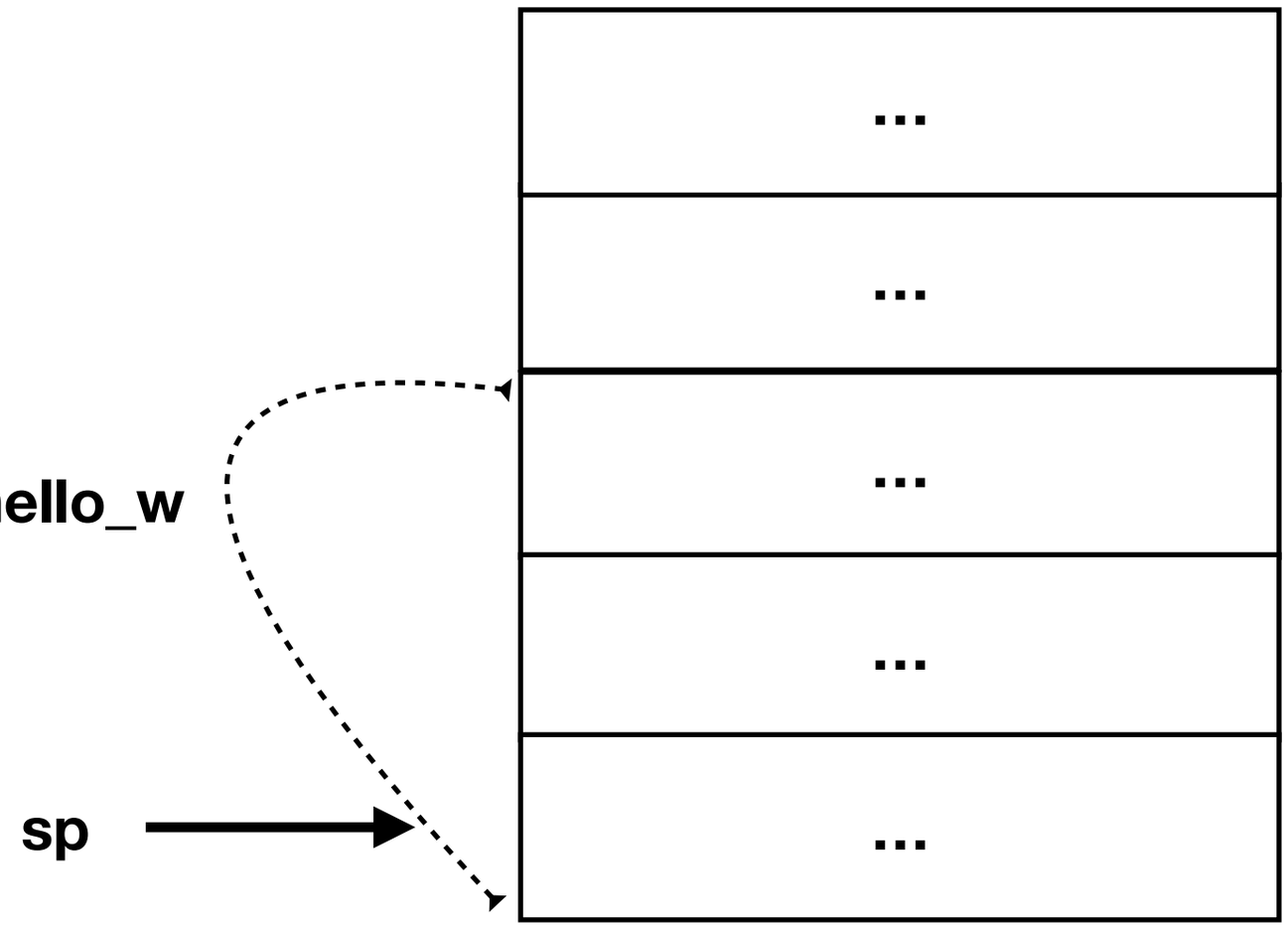
Thread Death

```
void hello_w() {
    printf(“Hello World!”);
    return;
}
```

- What happens after the return?

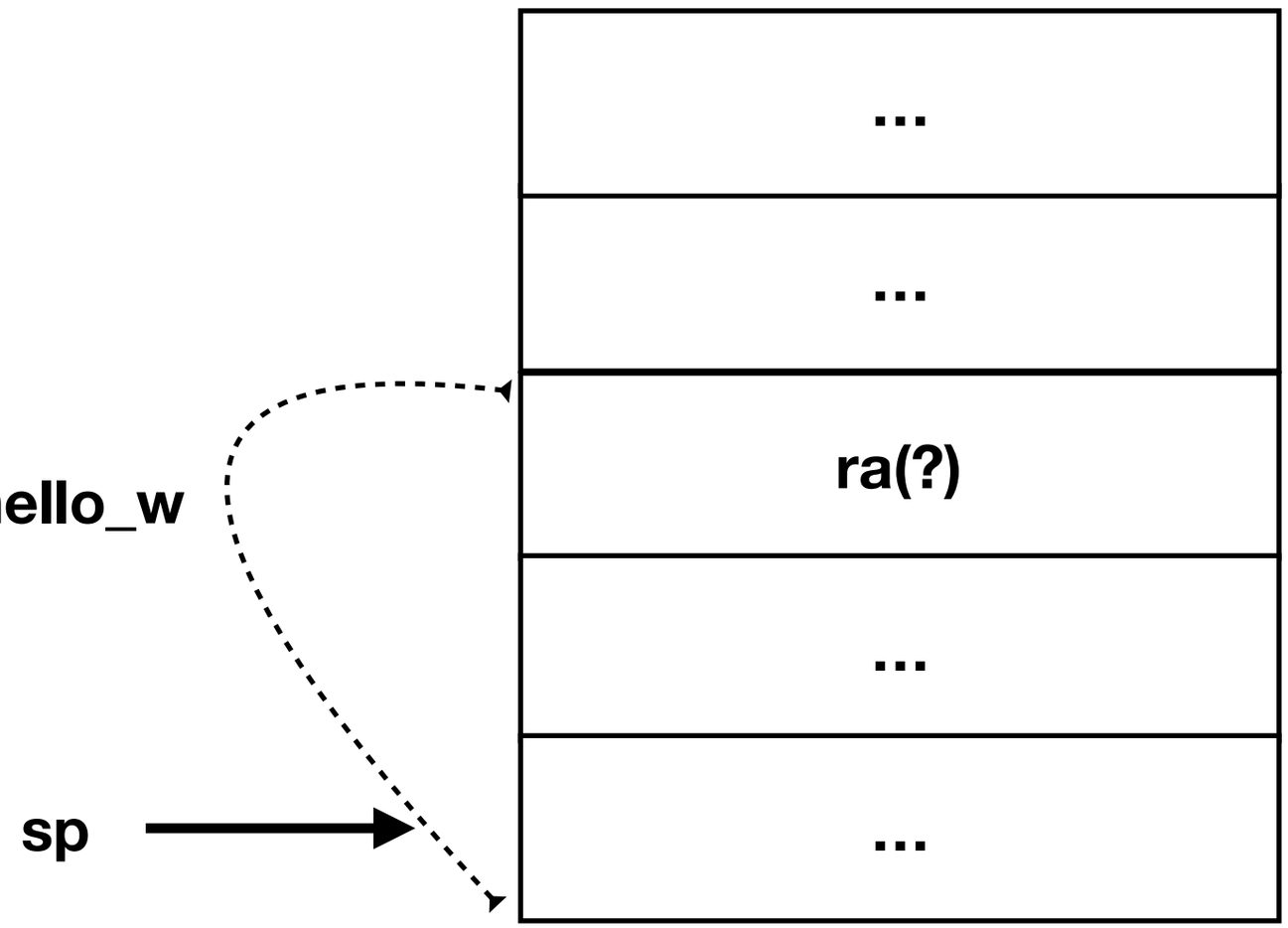
Thread Death

```
void hello_w() {
    printf("Hello World!");
    return;
}
```

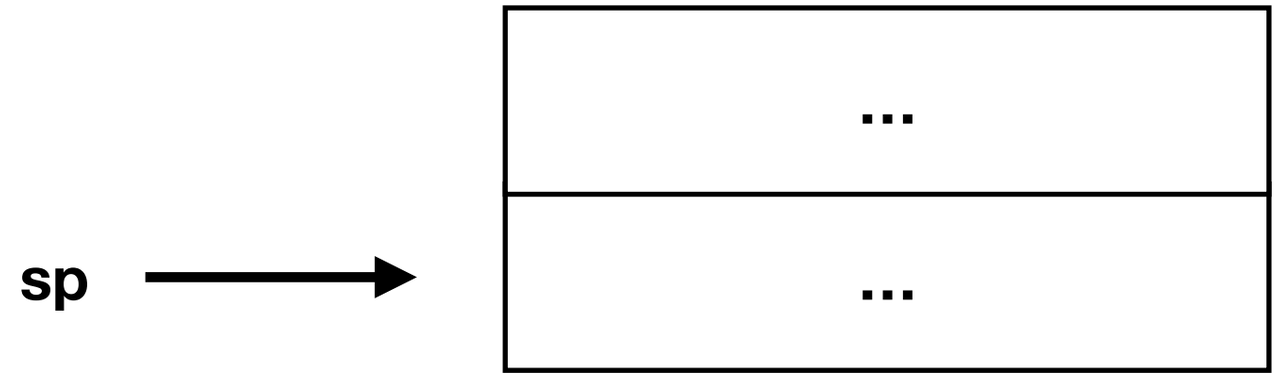


Thread Death

```
void hello_w() {
    printf("Hello World!");
    return;
}
```



Thread Death



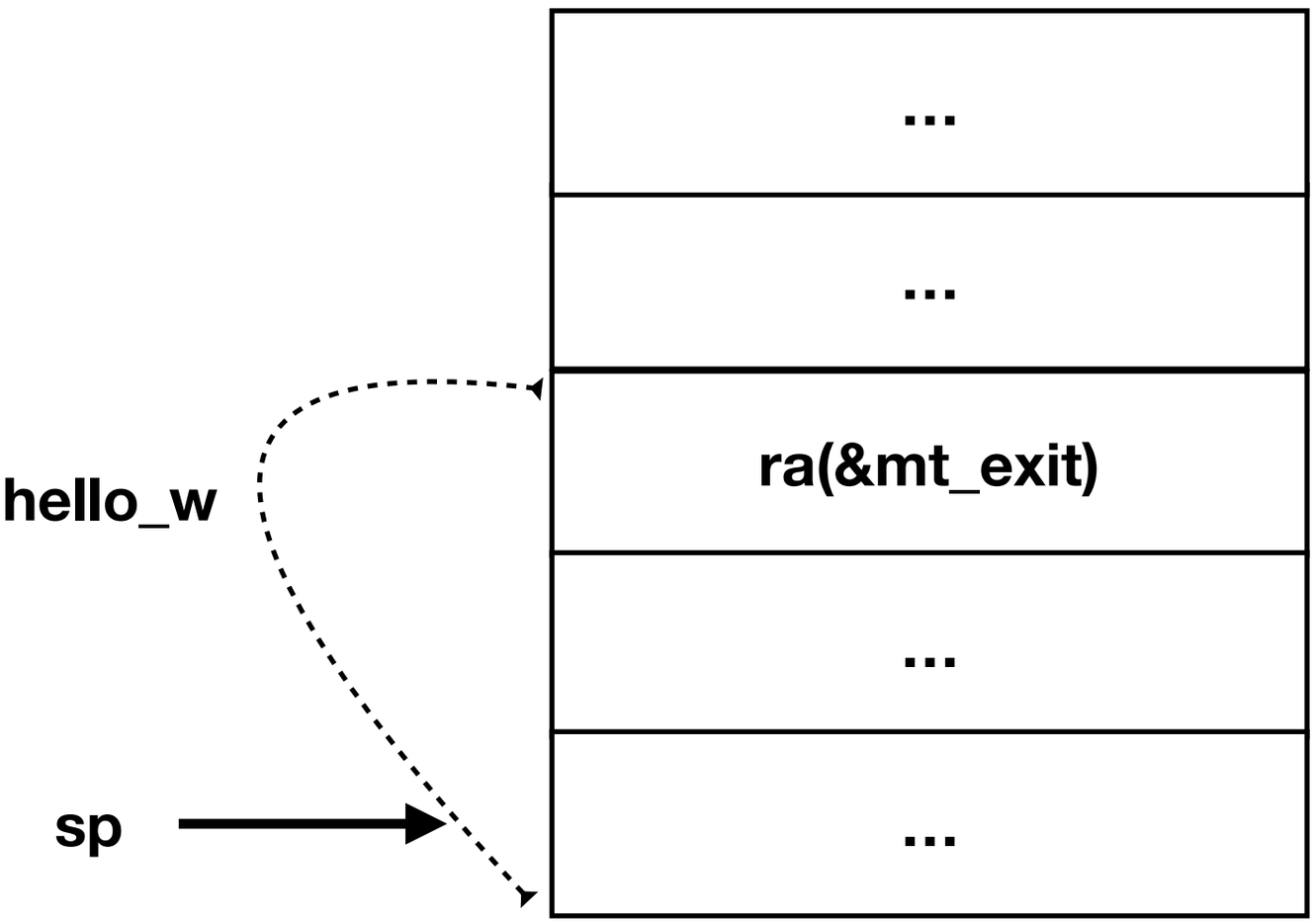

```
void hello_w() {
    printf(“Hello World!”);
    return;
}
```

pc = ?

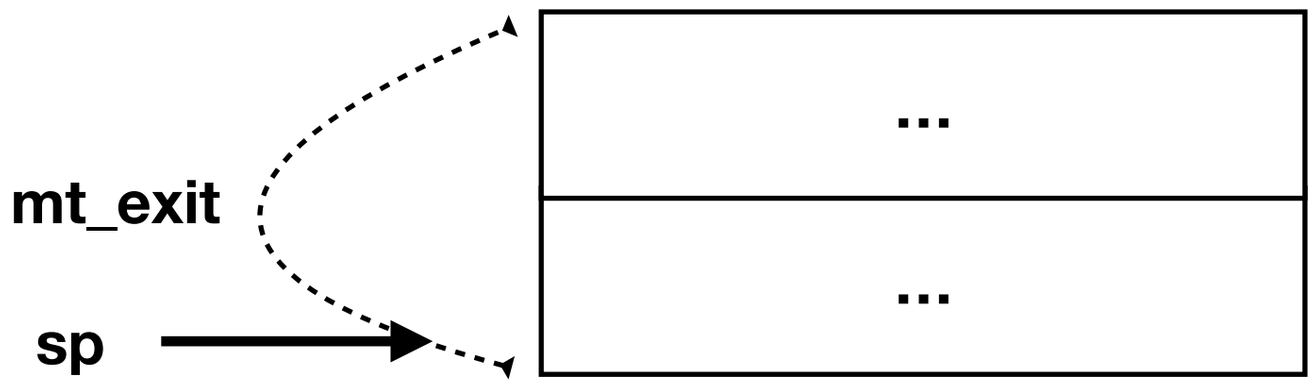
Thread Death

```
void hello_w() {
    printf("Hello World!");
    return;
}
```

```
void mt_exit() {
    //do cleanup
    while (1) {};
}
```



Thread Death




```
void hello_w() {
    printf("Hello World!");
    return;
}
```


```
void mt_exit() {
    //do cleanup
    while (1) {};
}
```

Thread Death

- Cleanup
  - Free stack
  - Free tcb
  - Any problems?
- Need another thread to do cleanup