P2: Cooperative Thread

Yu-Ju Huang

Robbert van Renesse

Implement the following interface:

void thread_init();

- initialize the user-level threading module (process becomes a thread)

void thread_create(void (*f)(void *arg), void *arg, unsigned int stack_size);

- create another thread that executes f(arg)

void thread_yield();

- yield to another thread (*thread scheduling is non-preemptive*)

void thread_exit();

- thread terminates and yields to another thread or terminates entire process

Example usage

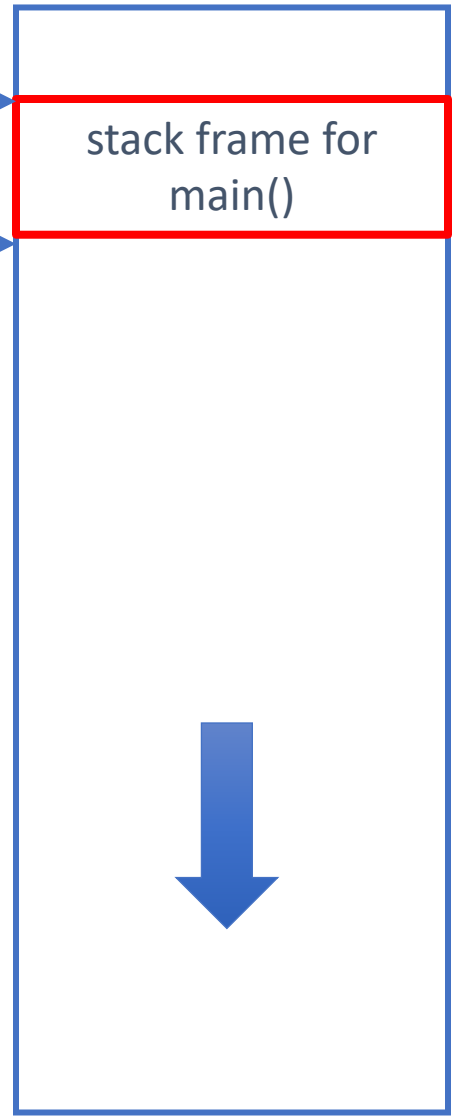
```
static void test_code(void *arg){
    int i;

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

int main(int argc, char **argv){
    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;
}
```

You’ll need to understand stacks *really well*

Review: stack (aka call stack)

```
int main(argc, argv){
        …
        f(3.14)
        …
}

int f(x){
        …
        g();
        …
}

int g(y){
        …
}
```

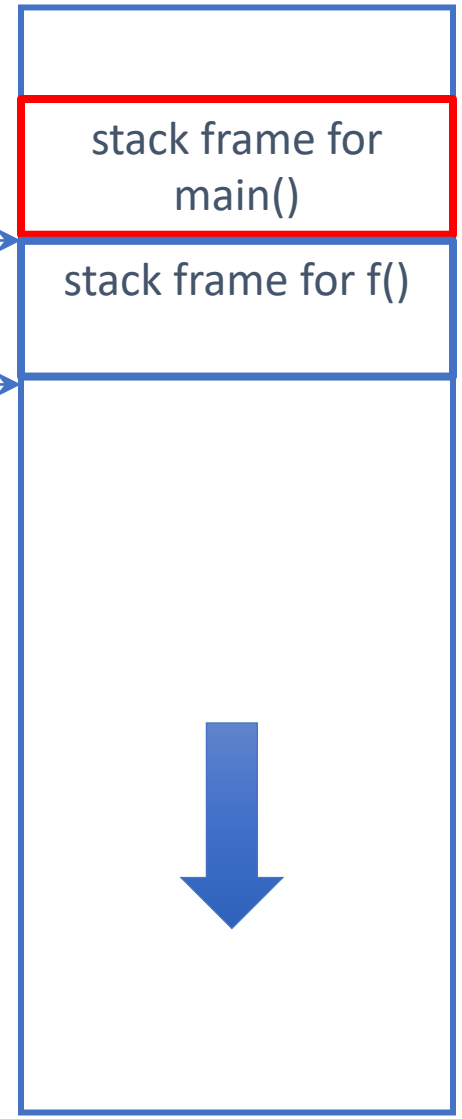
PC/IP →



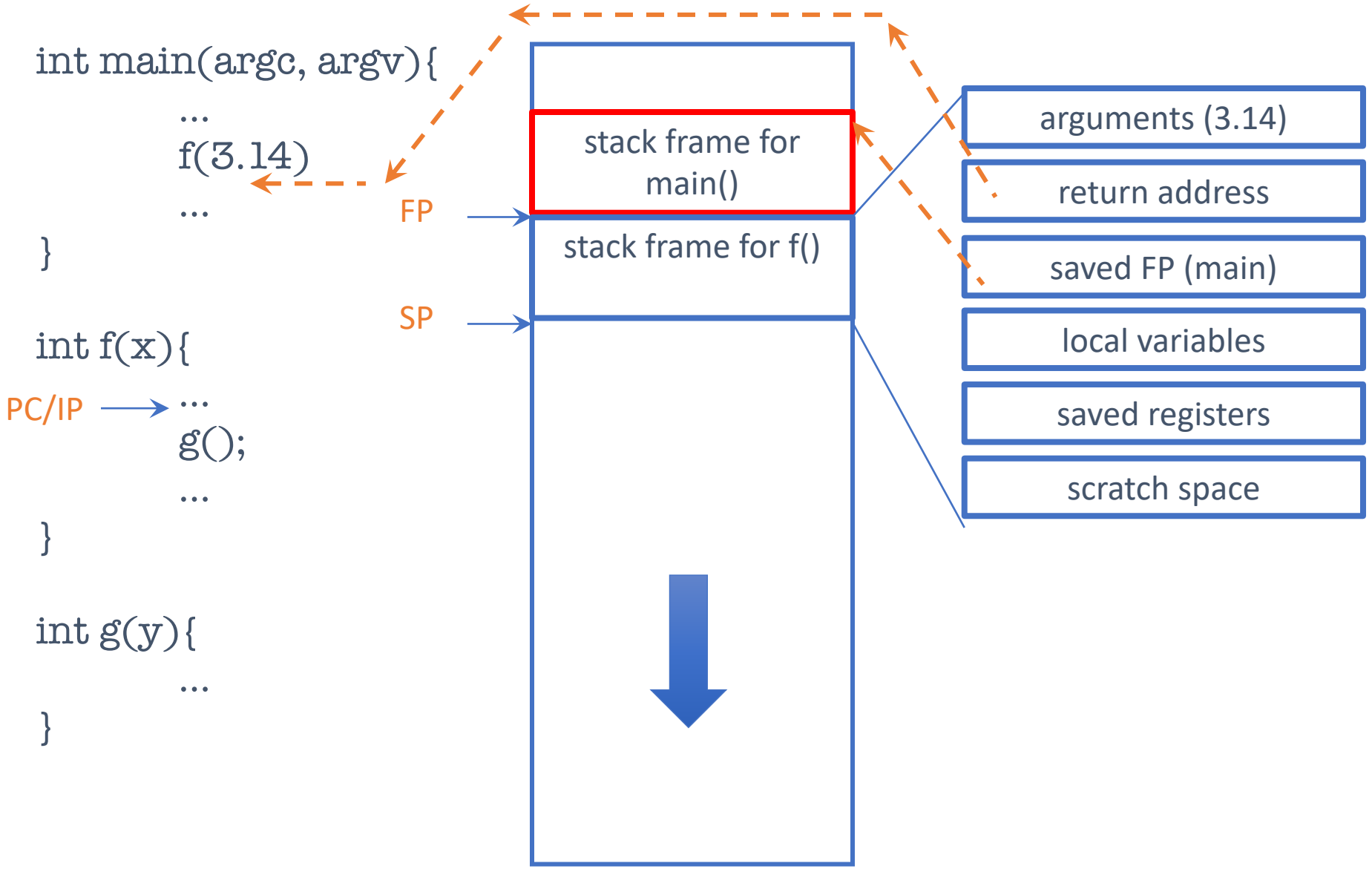
Review: stack (aka call stack)

```
int main(argc, argv){
        …
        f(3.14)
        …
}

int f(x){
        …
        g();
        …
}

int g(y){
        …
}
```

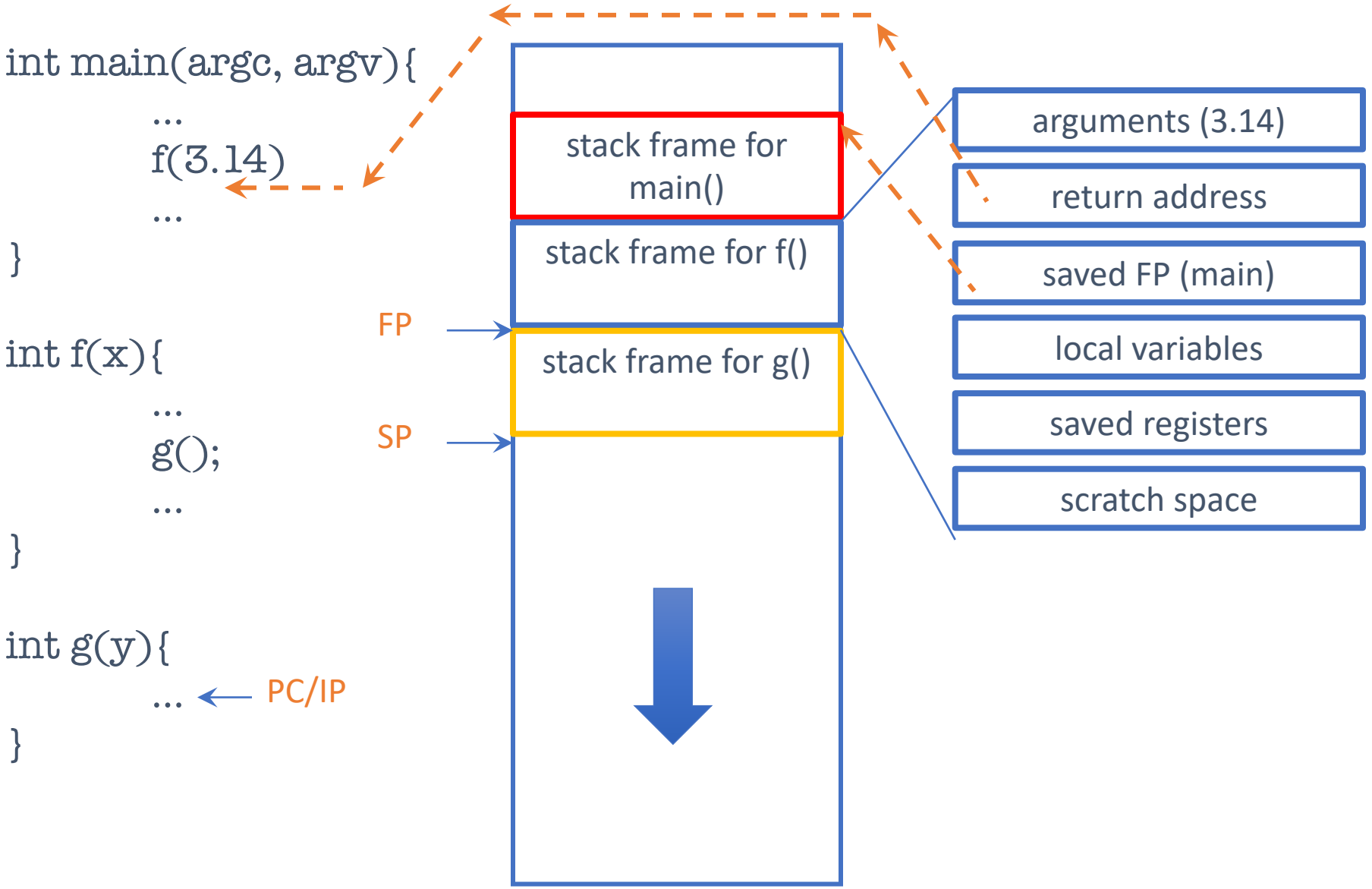
PC/IP →



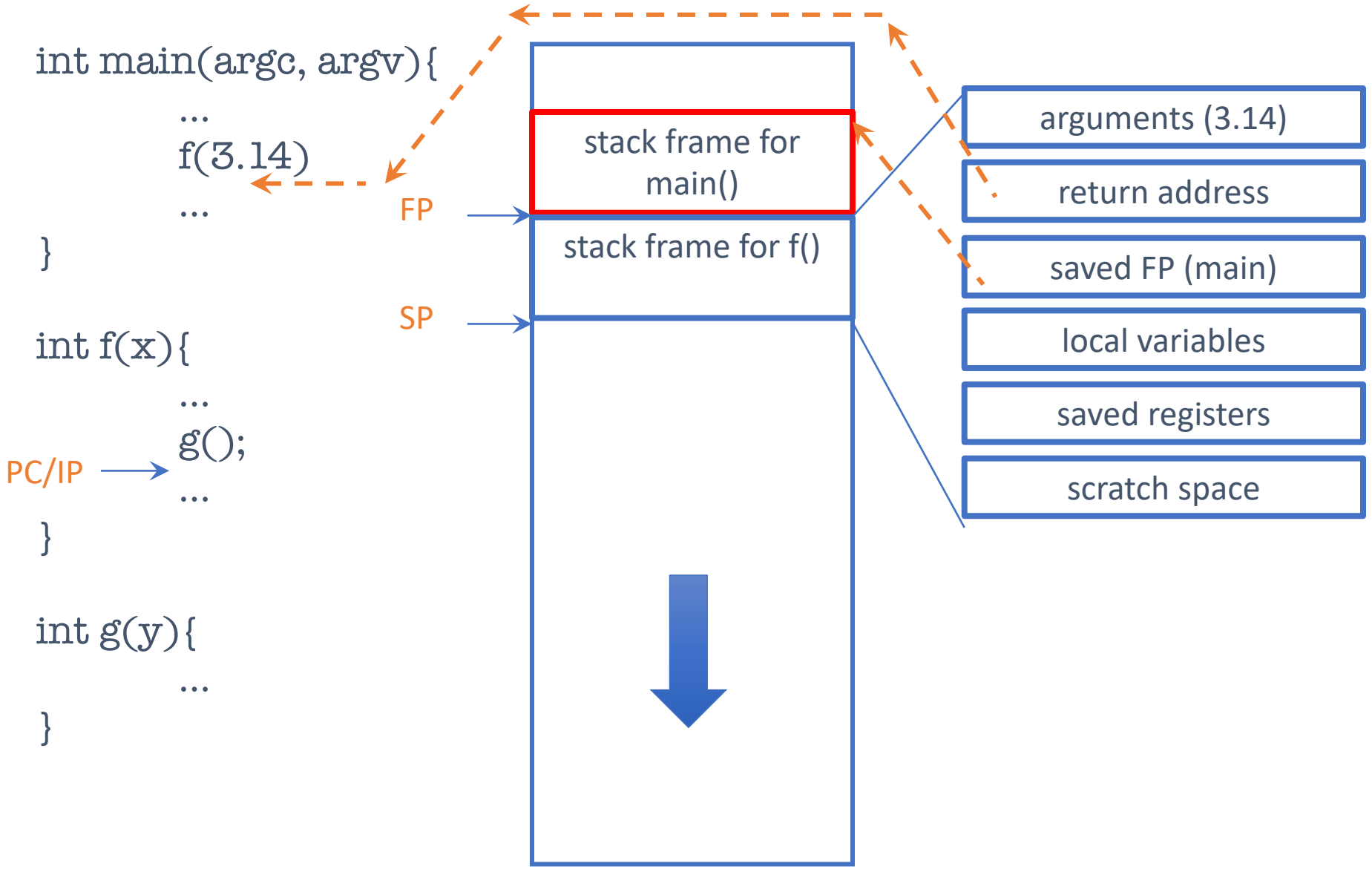
Review: stack (aka call stack)



Review: stack (aka call stack)



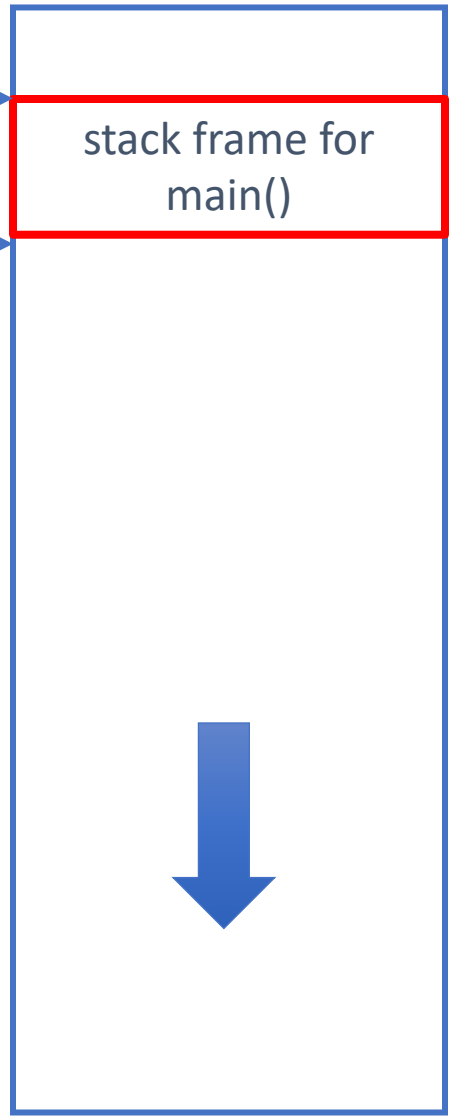
Review: stack (aka call stack)



Review: stack (aka call stack)

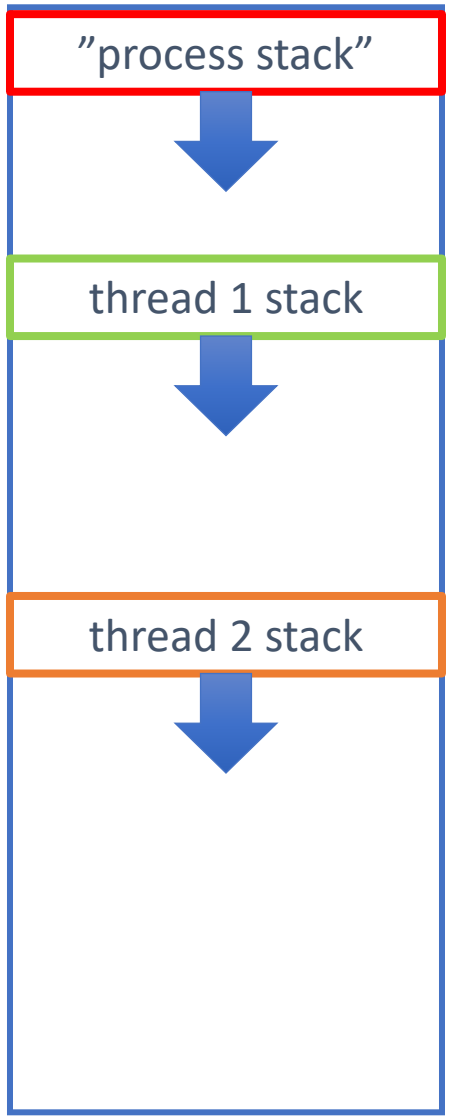
```
int main(argc, argv){
        ...
        f(3.14)
        ...
}

int f(x){
        ...
        g();
        ...
}

int g(y){
        ...
}
```

PC/IP



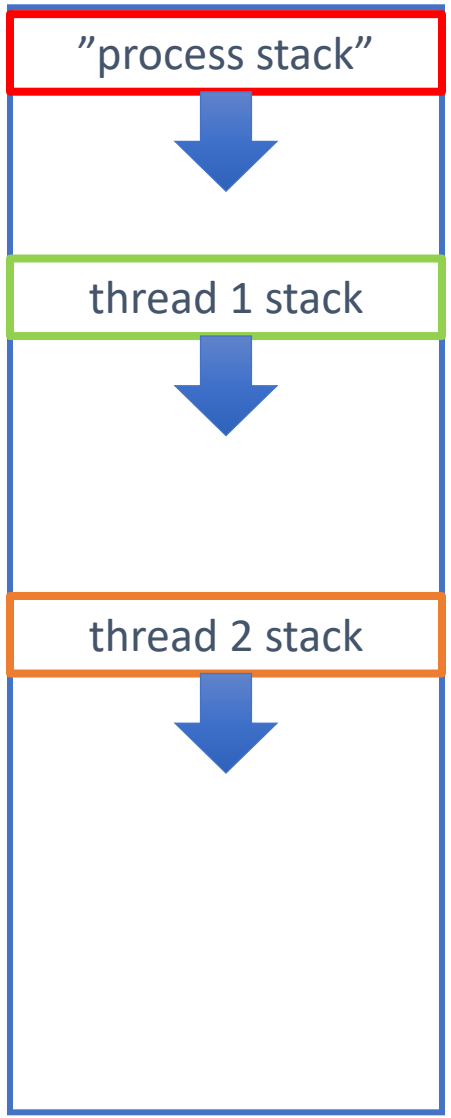
Each thread has its own stack!!

Each thread has its own stack!!



Each thread has its own stack!!

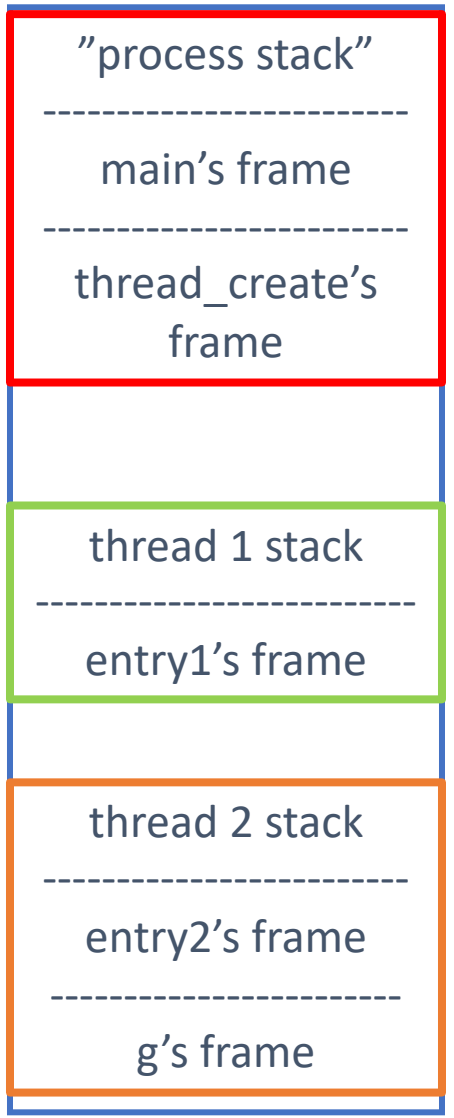
- And its own PC (aka IP), SP, FP, general purpose registers



Each thread has its own stack!!

- And its own PC (aka IP), SP, FP, general purpose registers
- And stack frames

```
void entry1(void* args) { ... }

void entry2(void* args) { g(); }

void g() { ... }

int main() {
   thread_init();
   thread_create(entry1, ...);
   thread_create(entry2, ...);
   ...
}
```



But we have only one CPU, one core

- And the process has only one stack

We need some magic…

(where’s the thread?)

We run one thread at a time

We run one thread at a time

Scheduling State of a Thread

- Running
  - currently running
- Runnable (aka Ready)
  - TCB on the run queue (aka ready queue)
- Terminated
  - TCB marked as having terminated

We run one thread at a time

- and save the state of the other threads in a secret location
- The state of a thread (aka *context*) consists of
  - its registers (including PC, SP, and FP)
  - its stack
  - possibly more stuff (scheduling state)

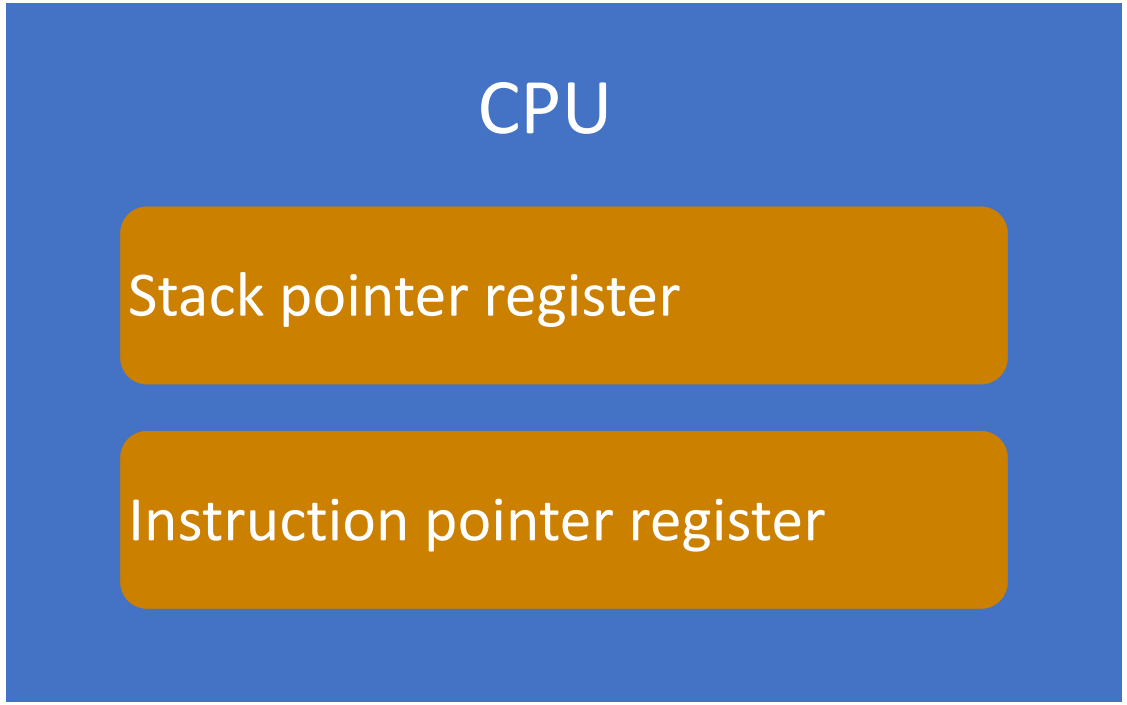
Context

- Context=
  - memory address space +
  - stack +
  - stack pointer +
  - instruction pointer or (program counter)
- At any moment, a CPU is in the context of some process

code & stack of both zoom and Keynote in memory

zoom stack **zoom** code **Keynote** code **Keynote** stack

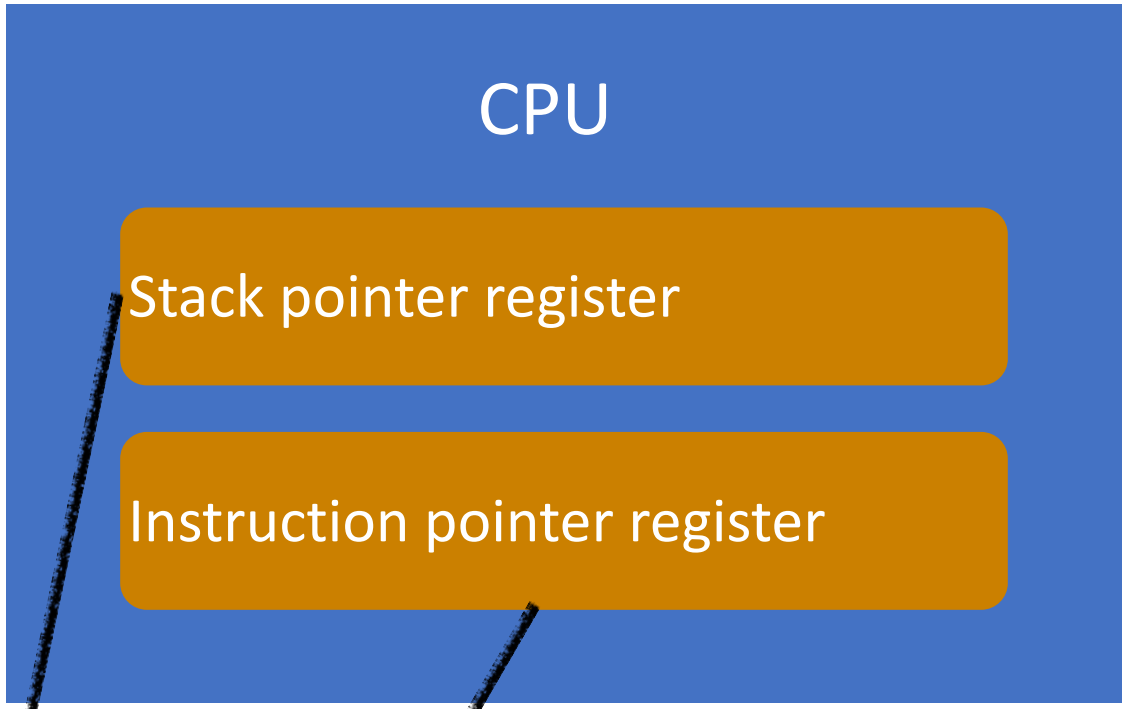
Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF



zoom stack | zoom code | Keynote code | Keynote stack

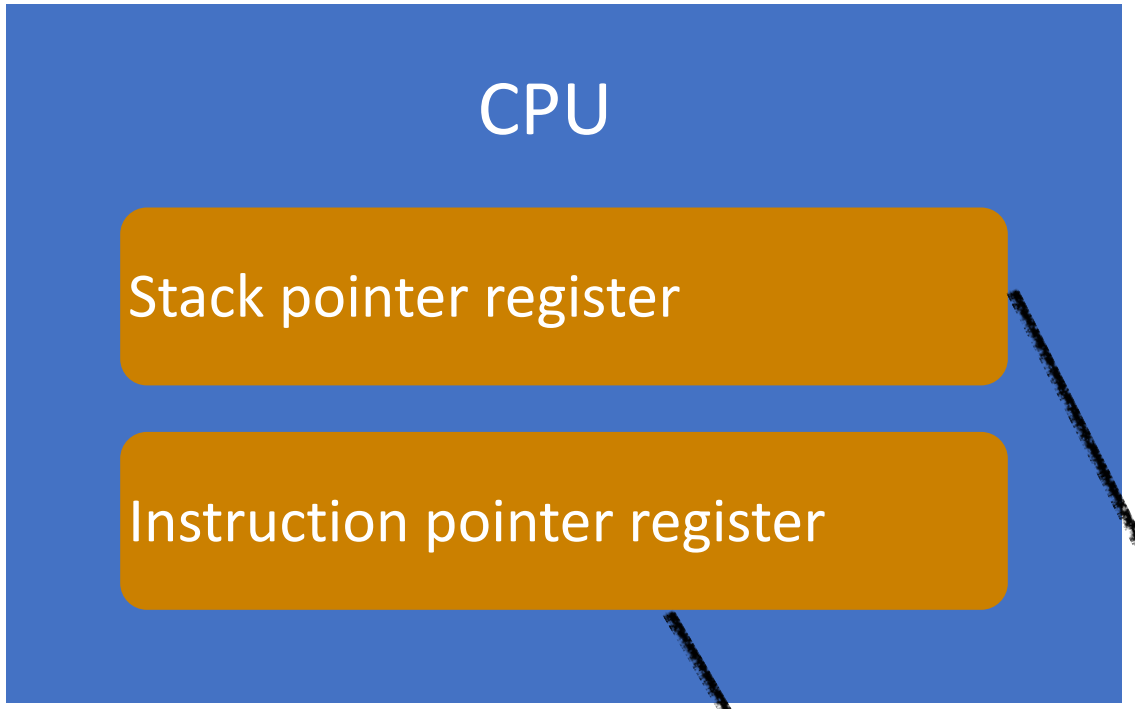
Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

CPU is in the context of zoom



zoom stack **zoom** code **Keynote** code **Keynote** stack

Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF



CPU is in the context of Keynote

zoom stack | **zoom** code | **Keynote** code | **Keynote** stack

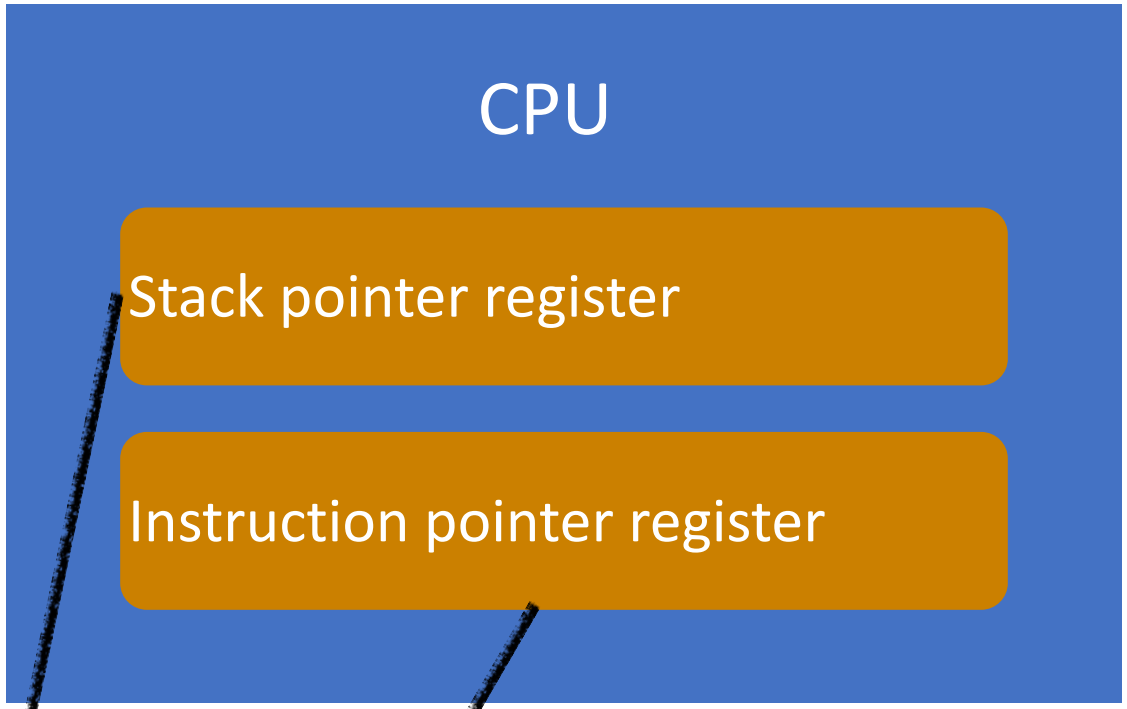
Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

Operating system is a program

zoom stack **zoom** code **OS** code **OS** stack

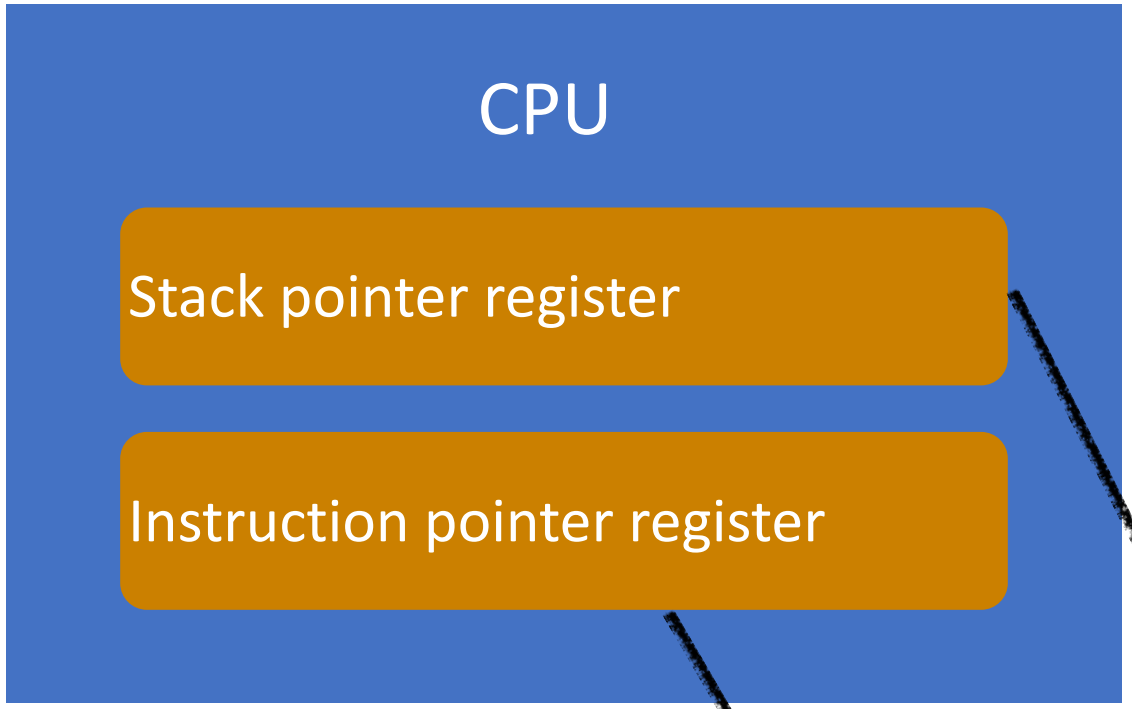
Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

zoom wants to use network, microphone, speaker, etc.



zoom stack **zoom** code **OS** code **OS** stack

Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF



OS manages the hardwares for zoom

zoom stack | **zoom** code | **OS** code | **OS** stack

Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

Context Switching

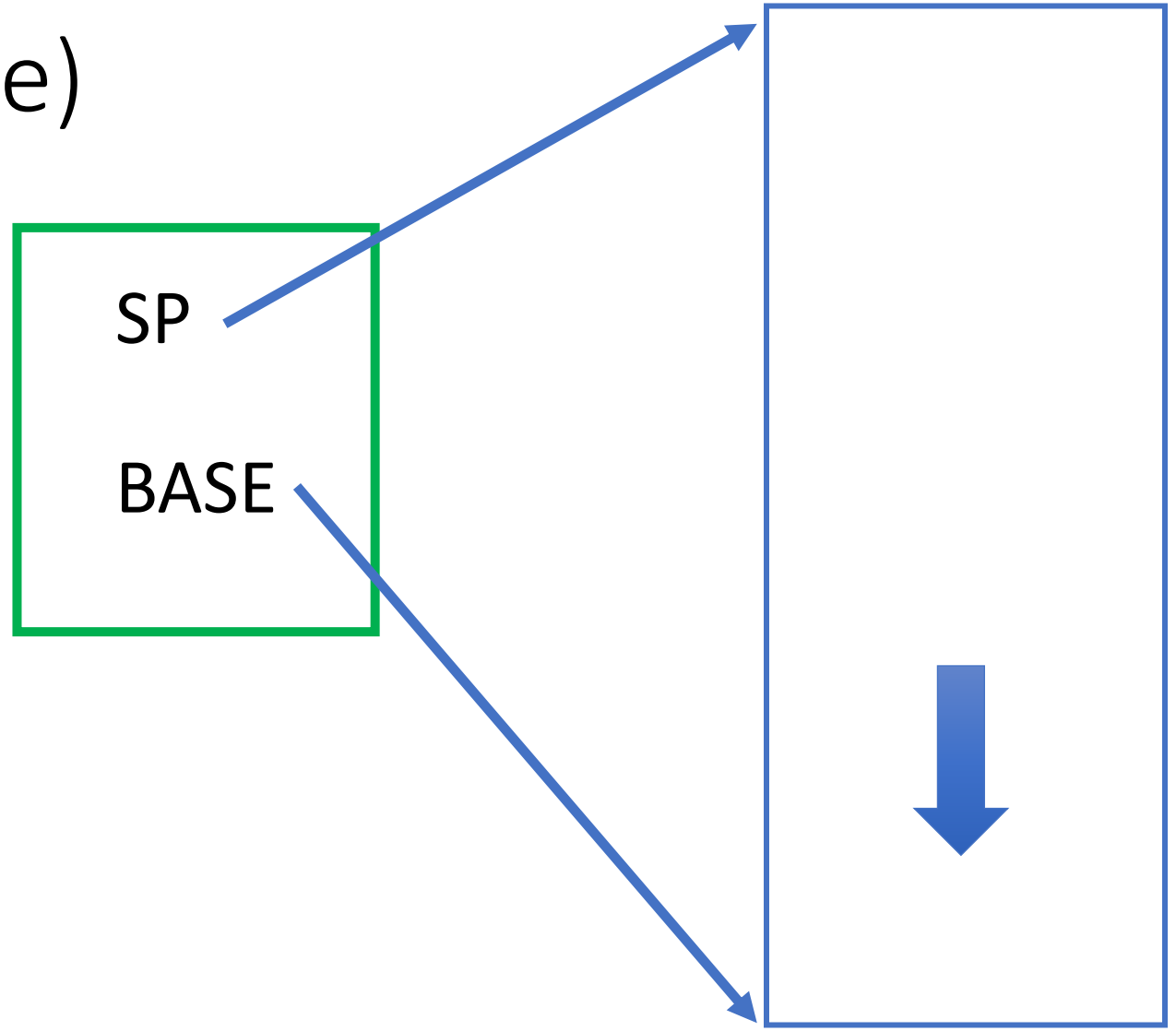
- When a thread exits (thread_exit) or yields (thread_yield) another thread, if any, gets to run
- If a thread yields, we need to save its context
- We need to be able to restore another context

Where to store the context of a thread?

- Convenient to push a thread's registers onto the stack
  - but you can't use the stack to find stack pointer…

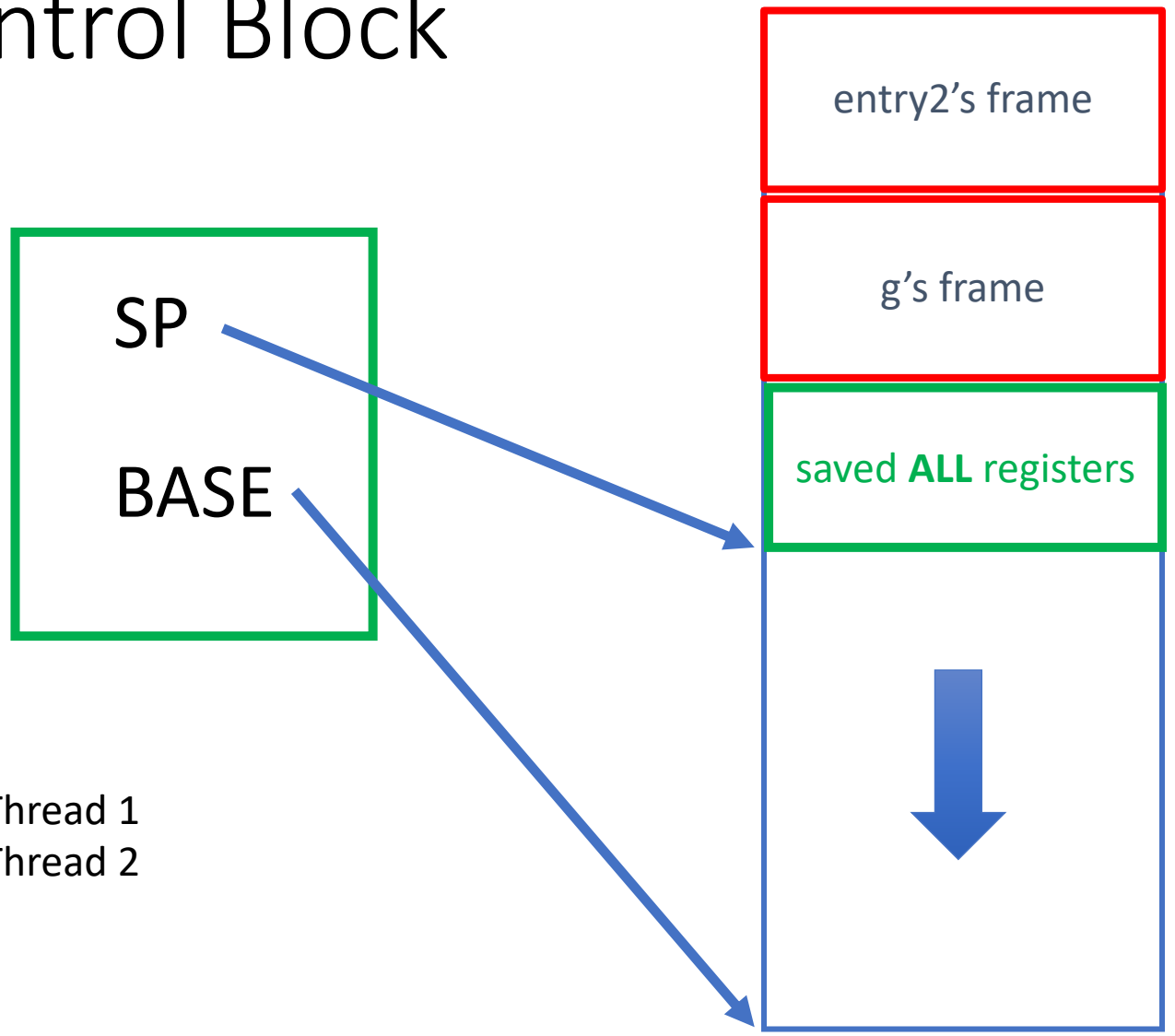
- Keep the stack pointer in a ***Thread Control Block***
  - one TCB per thread

Thread Control Block (initial state)



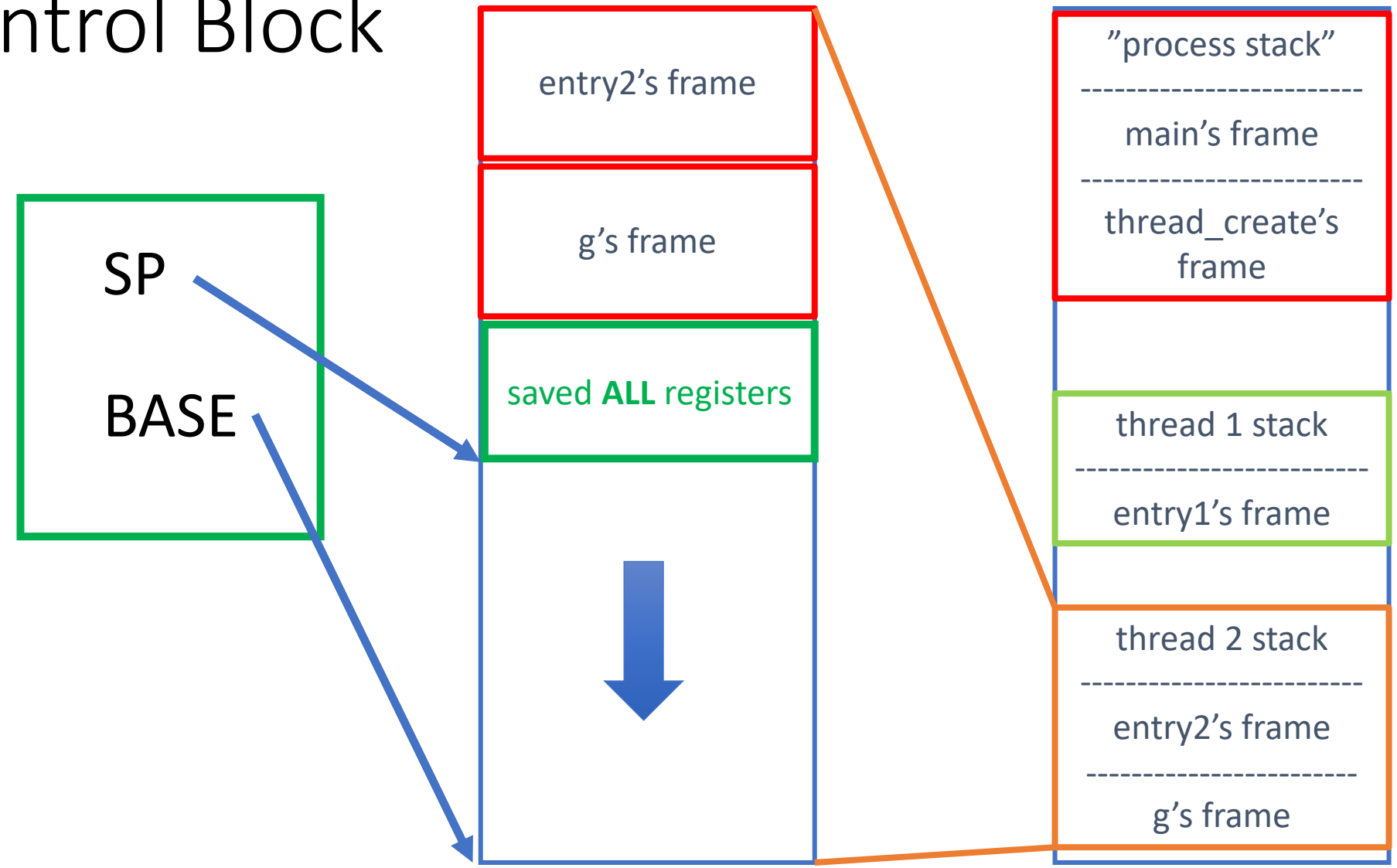
Thread Control Block

```
void entry1(void* args) { ... }

void entry2(void* args) { g(); }

void g() { thread_yield(); }

int main() {
   thread_init();
   thread_create(entry1, ...);  // Thread 1
   thread_create(entry2, ...);  // Thread 2
   ...
}
```



Thread Control Block

```
void entry1(void* args) { ... }

void entry2(void* args) { g(); }

void g() { thread_yield(); }

int main() {
   thread_init();
   thread_create(entry1, ...);
   thread_create(entry2, ...);
   ...
}
```



Data Structures

- Thread

- Queue

Data Structures

- Thread
  - Function
  - Arguments
  - State
  - Stack memory
  - Stack pointer
- Queue
  - RUNNABLE Threads

```
ctx_switch:
    addi sp,sp,-128
    SAVE_ALL_REGISTERS
    sw sp,0(a0)
    mv sp,a1
    RESTORE_ALL_REGISTERS
    addi sp,sp,128
    ret
```

```
void thread_yield(){
        ……
        next = runQueue.pop();
        next->state = RUNNING;
        ctx_switch(&current->sp, next->sp);
        ……
}
```

```
.macro SAVE_ALL_REGISTERS
    sw ra, 0(sp)
    sw t0, 4(sp)
    sw t1, 8(sp)
      ……
    sw s10, 104(sp)
    sw s11, 108(sp)
    sw gp,  112(sp)
    sw tp,  116(sp)
    sw sp,  120(sp)
.endm
```

```
.macro RESTORE_ALL_REGISTERS
    lw ra, 0(sp)
    lw t0, 4(sp)
    lw t1, 8(sp)
      ……
    lw s10, 104(sp)
    lw s11, 108(sp)
    lw gp,  112(sp)
    lw tp,  116(sp)
    lw sp,  120(sp)
.endm
```

```
ctx_switch:
    addi sp,sp,-128
    SAVE_ALL_REGISTERS
    sw sp,0(a0)
    mv sp,a1
    RESTORE_ALL_REGISTERS
    addi sp,sp,128
    ret
```

```
void thread_yield(){
        ……
        next = runQueue.pop();
        next->state = RUNNING;
        ctx_switch(&current->sp, next->sp);
        ……
}
```

sp: stack pointer

RISC-V Calling Convention:

- **a0**: First argument
- **a1**: Second argument

```
.macro SAVE_ALL_REGISTERS
    sw ra, 0(sp)
    sw t0, 4(sp)
    sw t1, 8(sp)
      ……
    sw s10, 104(sp)
    sw s11, 108(sp)
    sw gp,  112(sp)
    sw tp,  116(sp)
    sw sp,  120(sp)
.endm
```

```
.macro RESTORE_ALL_REGISTERS
    lw ra, 0(sp)
    lw t0, 4(sp)
    lw t1, 8(sp)
      ……
    lw s10, 104(sp)
    lw s11, 108(sp)
    lw gp,  112(sp)
    lw tp,  116(sp)
    lw sp,  120(sp)
.endm
```

```
ctx_start:
    addi sp,sp,-128
    SAVE_ALL_REGISTERS
    sw sp,0(a0)
    mv sp,a1
    call ctx_entry
```

```
void thread_create(){
        ……
        current->state = RUNNABLE;
        runQueue.add(current);
        next = CREATE_NEW_TBC();
        ctx_start(&current->sp, next's top_of_stack);
        ……
}

Void ctx_entry() { … }
```

```
.macro SAVE_ALL_REGISTERS
    sw ra, 0(sp)
    sw t0, 4(sp)
    sw t1, 8(sp)
      ……
    sw s10, 104(sp)
    sw s11, 108(sp)
    sw gp,  112(sp)
    sw tp,  116(sp)
    sw sp,  120(sp)
.endm
```

```
.macro RESTORE_ALL_REGISTERS
    lw ra, 0(sp)
    lw t0, 4(sp)
    lw t1, 8(sp)
      ……
    lw s10, 104(sp)
    lw s11, 108(sp)
    lw gp,  112(sp)
    lw tp,  116(sp)
    lw sp,  120(sp)
.endm
```

thread_init()

- Initializes thread package
- Maintains *run queue* and *current thread*
- Allocates a TCB, but *not* a stack
  - because process already has one in use
- Set TCB->*base* to NULL to mark no stack has been allocated
- Initial run queue is empty
- Current thread points to allocated TCB

thread_create(f, arg, stack_size)

- Create a new thread
- Allocates a TCB and a stack (of the given size)
  - set TCB->*base* to “bottom”, and TCB->*sp* to “top”
- May or may not immediately switch to the new thread
  - I think it’s easier if you switch immediately

thread_yield()

- See if the run queue is empty
  - if so, we're done
- Get next TCB of the run queue
- Put current TCB on the run queue
- **Switch contexts**
  - Save registers on the stack
  - Save sp in current TCB
  - Restore sp of next TCB
  - Restore registers from the stack

thread_exit()

- See if the run queue is empty
  - if so, exit from the process using exit(0)
- Mark TERMINATED in TCB
- Get next TCB of the run queue
- **Switch contexts**
  - Save registers on the stack
  - Save sp in current TCB
  - Restore sp of next TCB
  - Restore registers from the stack
- Next thread cleans up last thread

Today

- Threading library
- P0 (Queue) due
- P1 (HelloWorld) due next week

- P2 (Thread) release, due in three weeks (Feb 28th)