# P1: Implement a Multi-Threading Package (in user space)

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

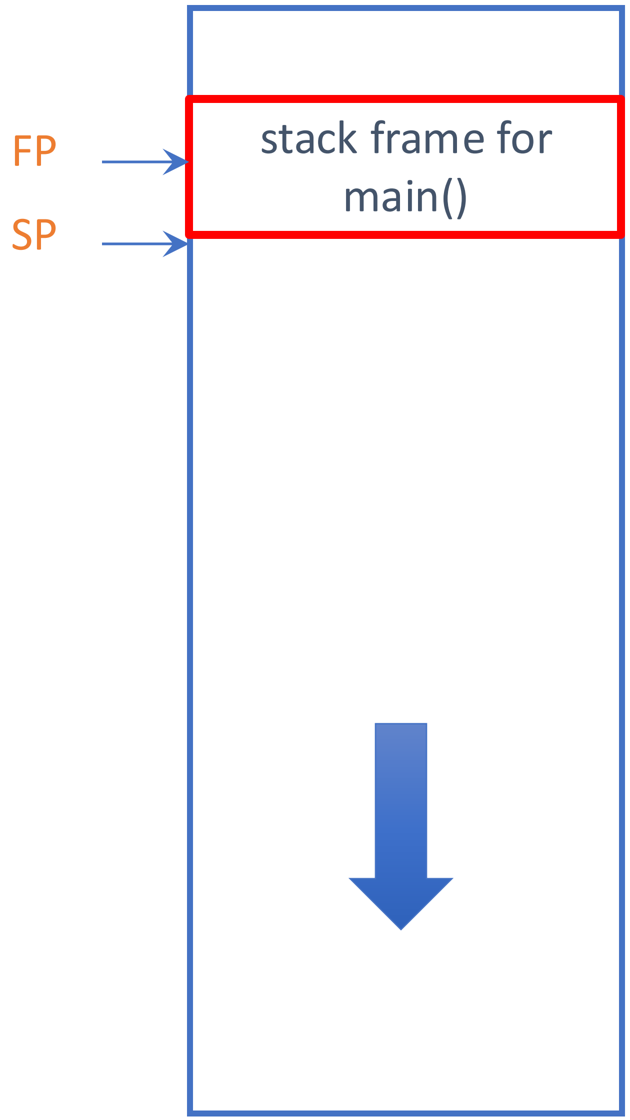
```
static void test_code(void *arg){
    int i;

    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)

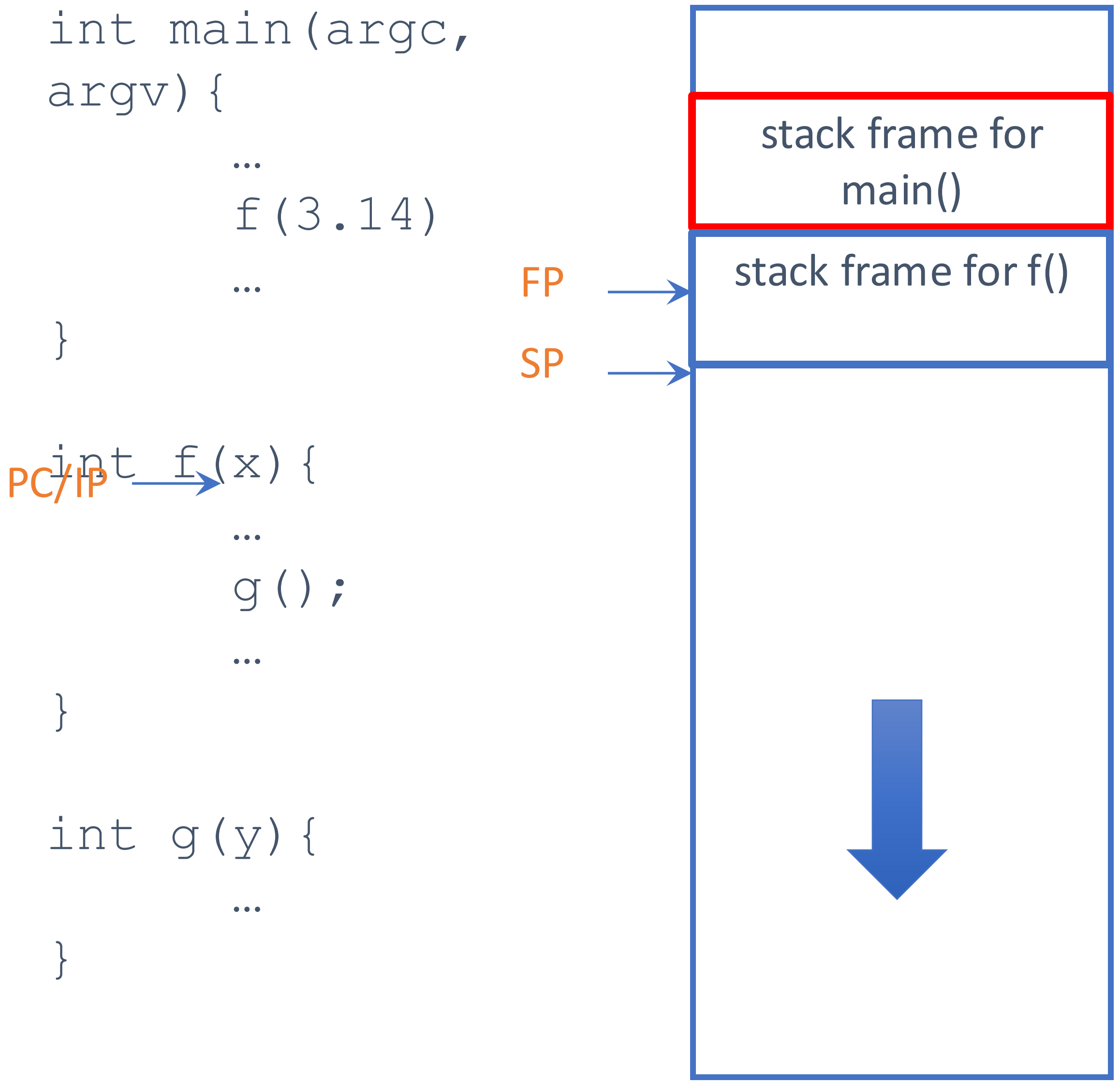
PC/IP

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

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

int g(y){
        …
}
```



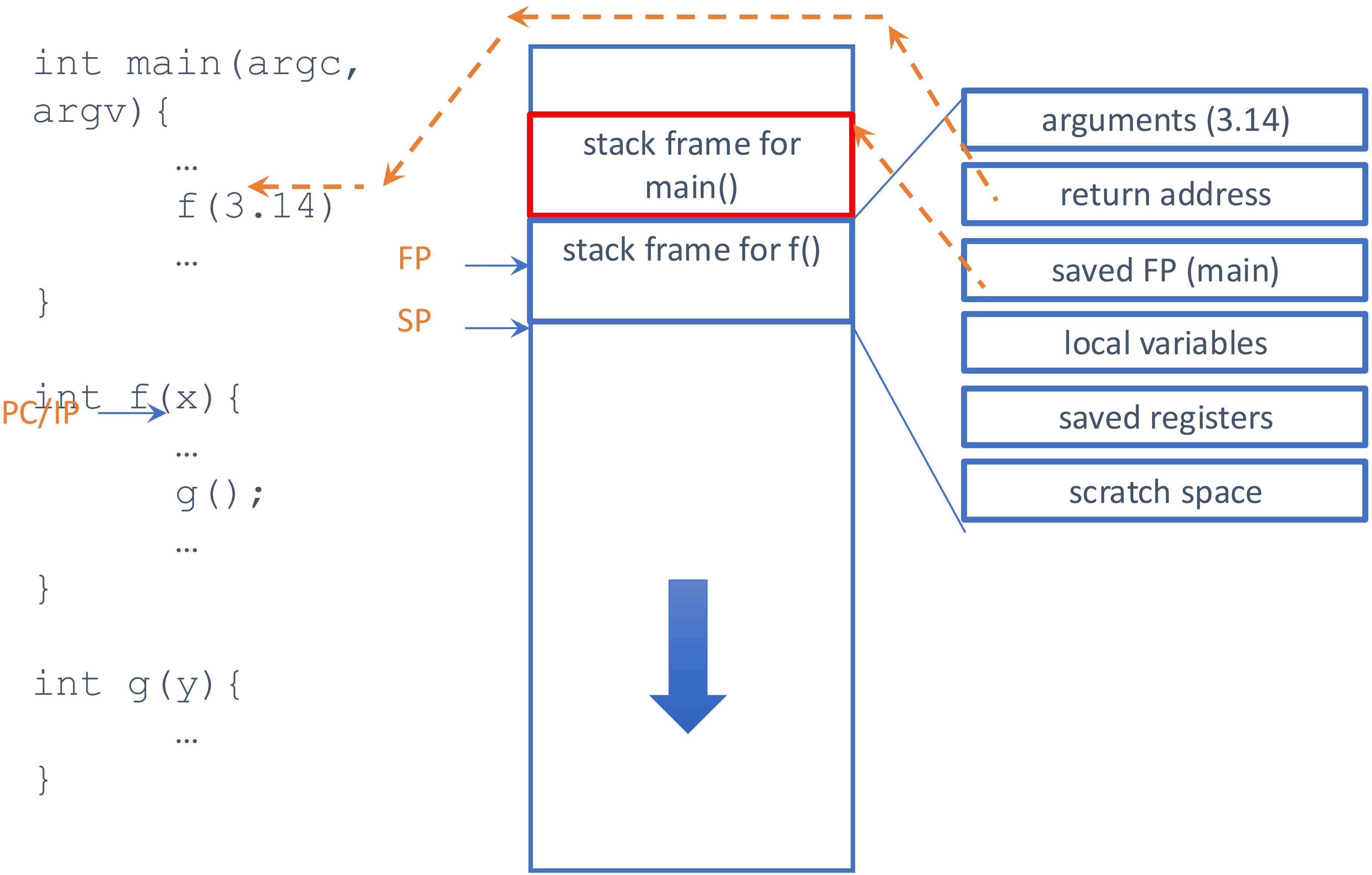
# Review: stack (aka call stack)

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

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

int g(y){
        …
}
```



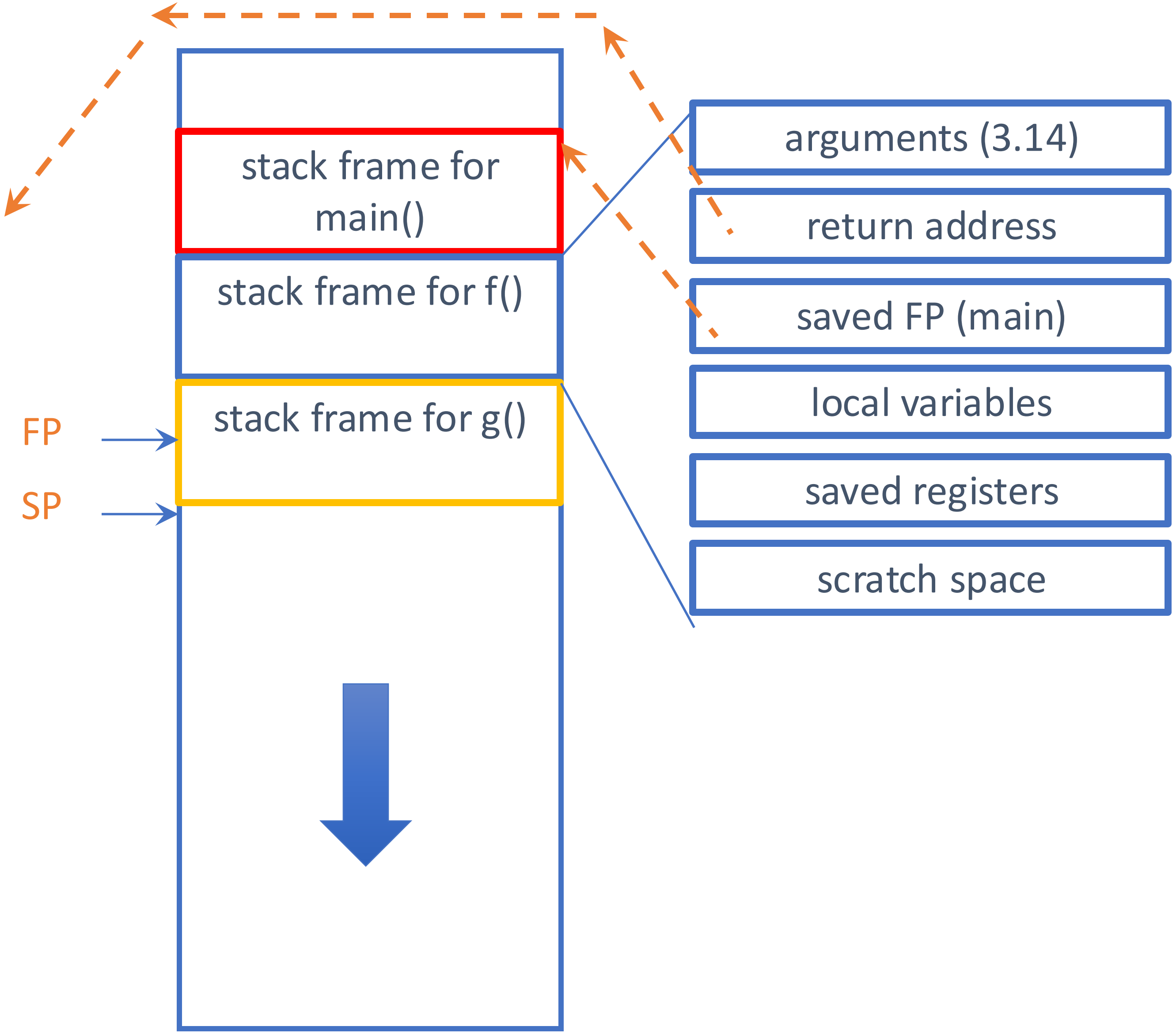
# Review: stack (aka call stack)

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

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

int g(y){
        …
}
```



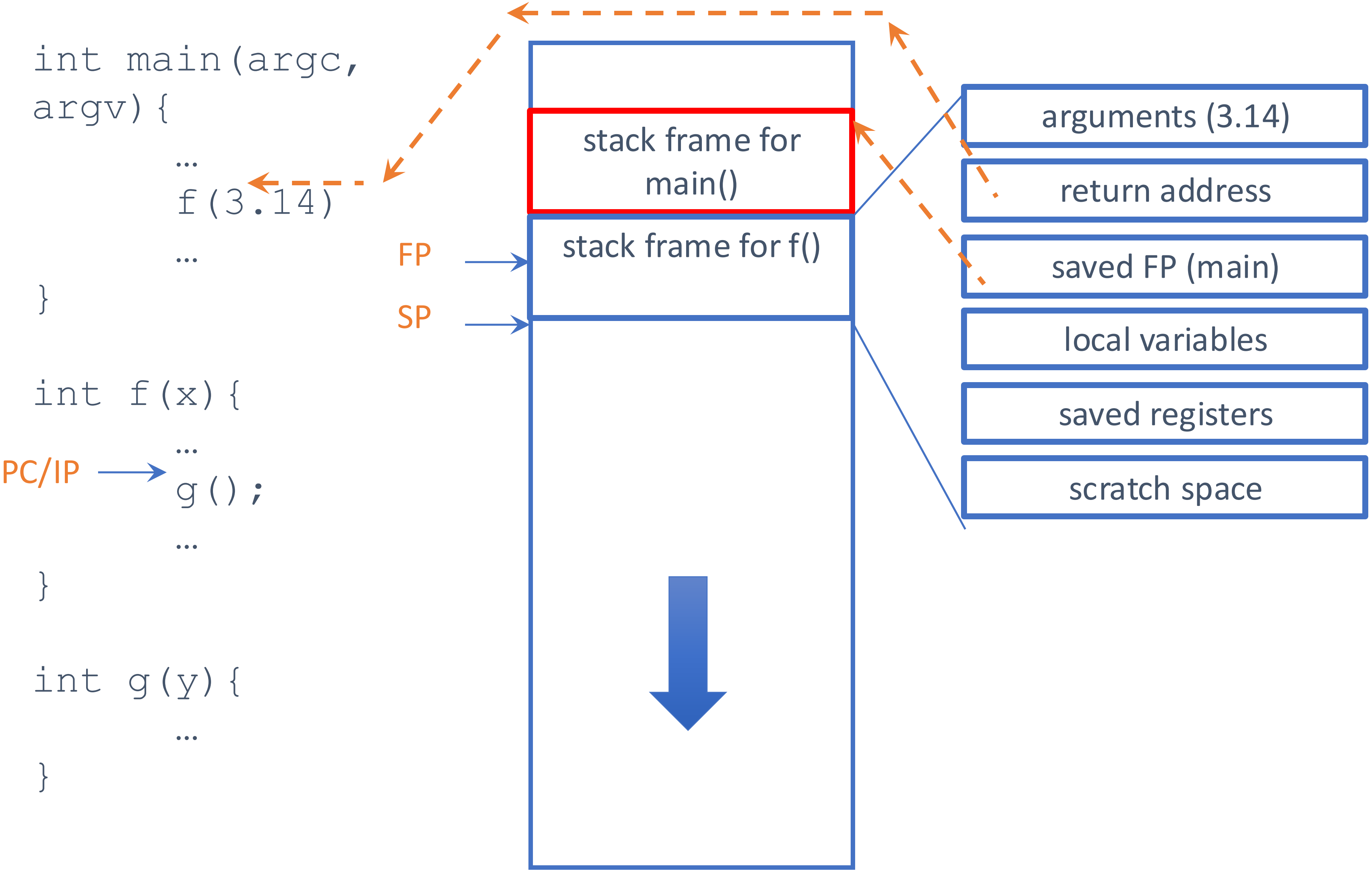
# Review: stack (aka call stack)

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

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

int g(y){ ← PC/IP
        …
}
```



# Review: stack (aka call stack)

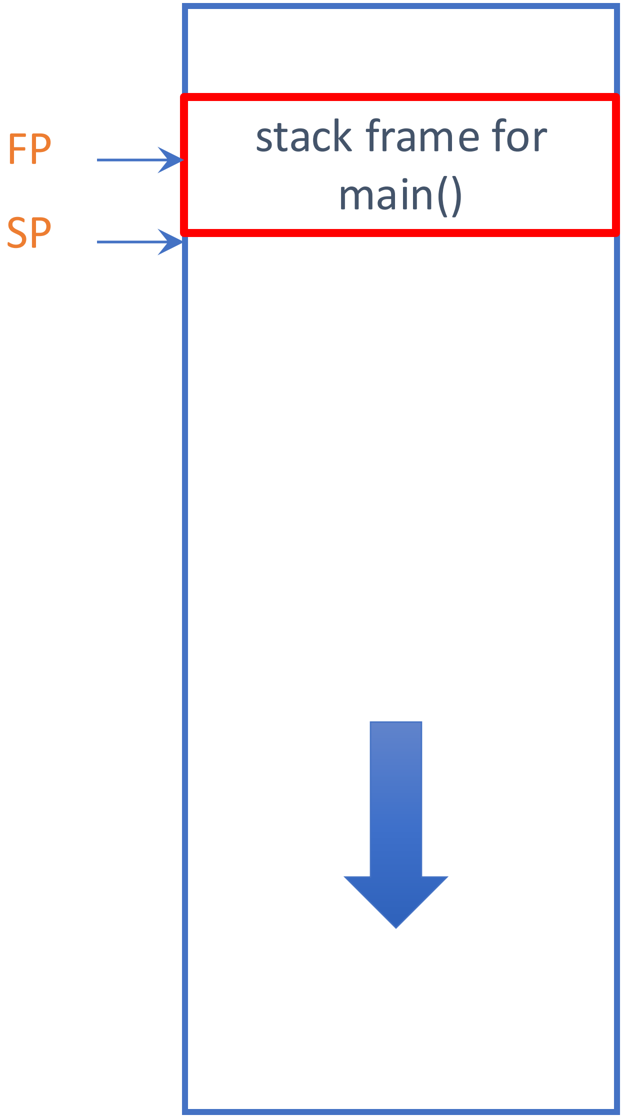
```
int main(argc, argv){
        …
        f(3.14)
        …
}

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

int g(y){
        …
}
```



# Review: stack (aka call stack)

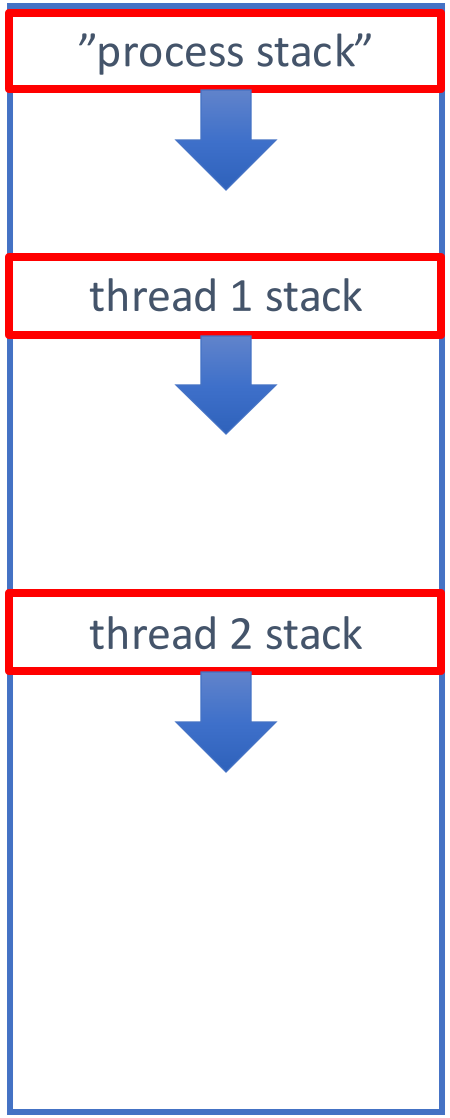
```
int main(argc, argv){
        …
        f(3.14)
        …
}

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

int g(y){
        …
}
```

PC/IP →



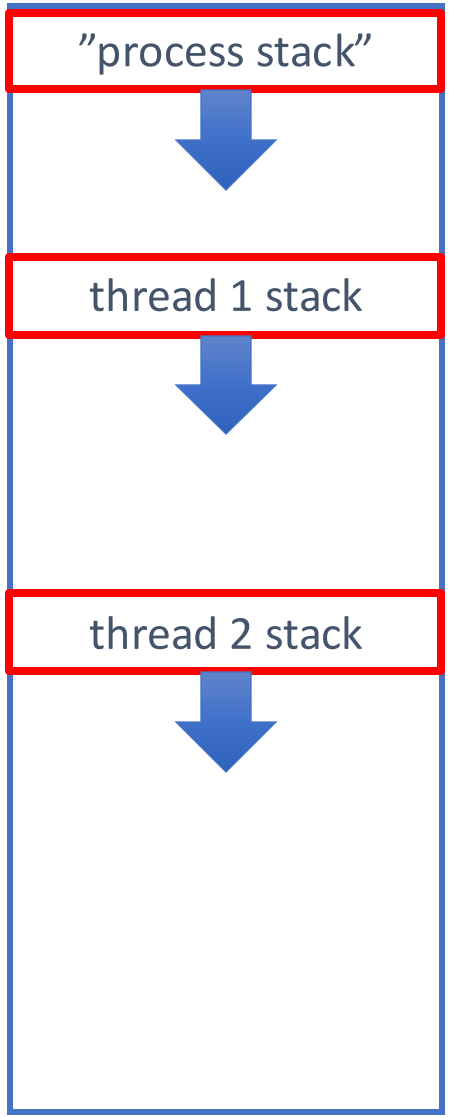
# Each thread has its own stack!!

# Each thread has its own stack!!



# Each thread has its own stack!!

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



# 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

- 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)

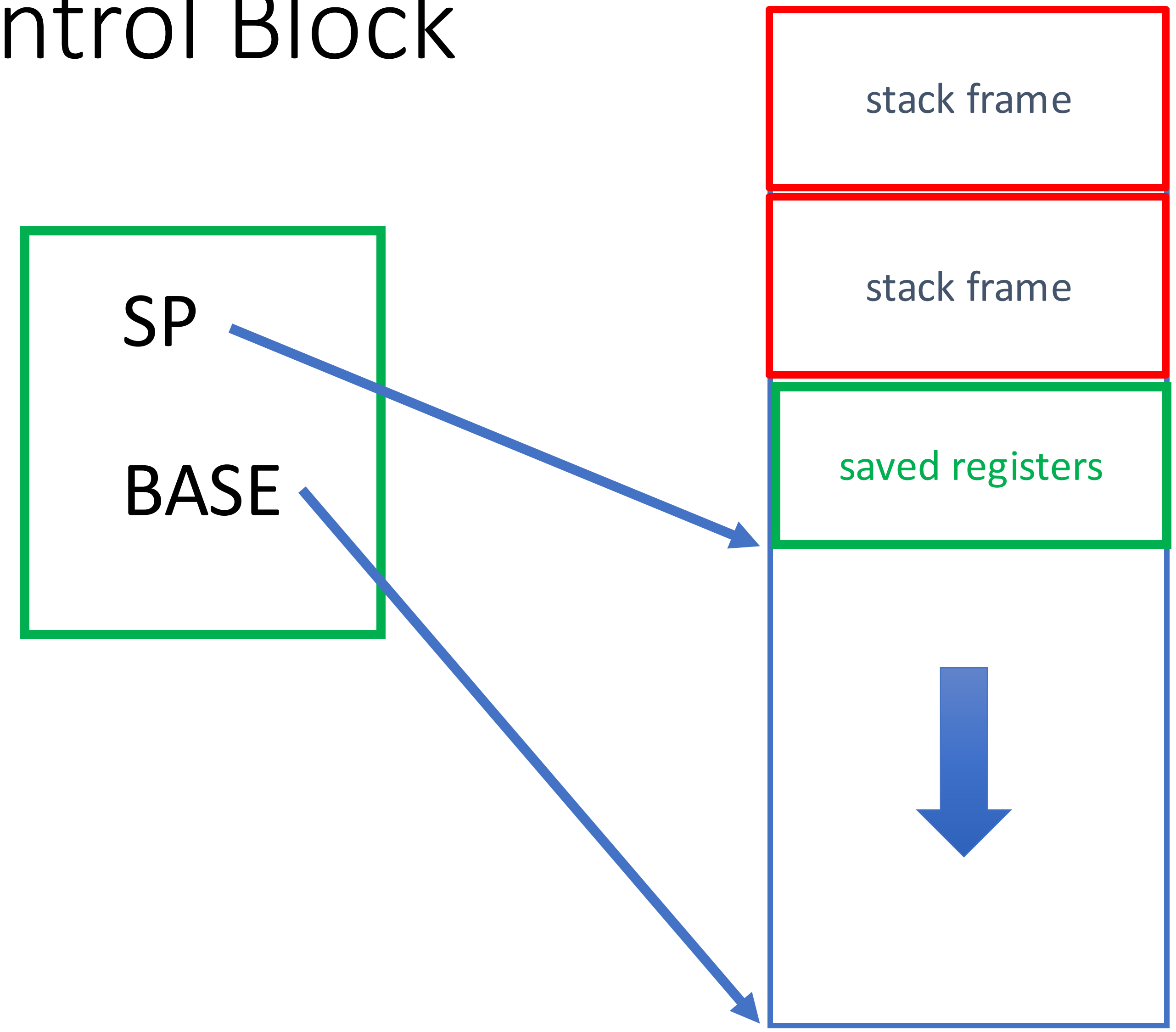
# Context Switching

- 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

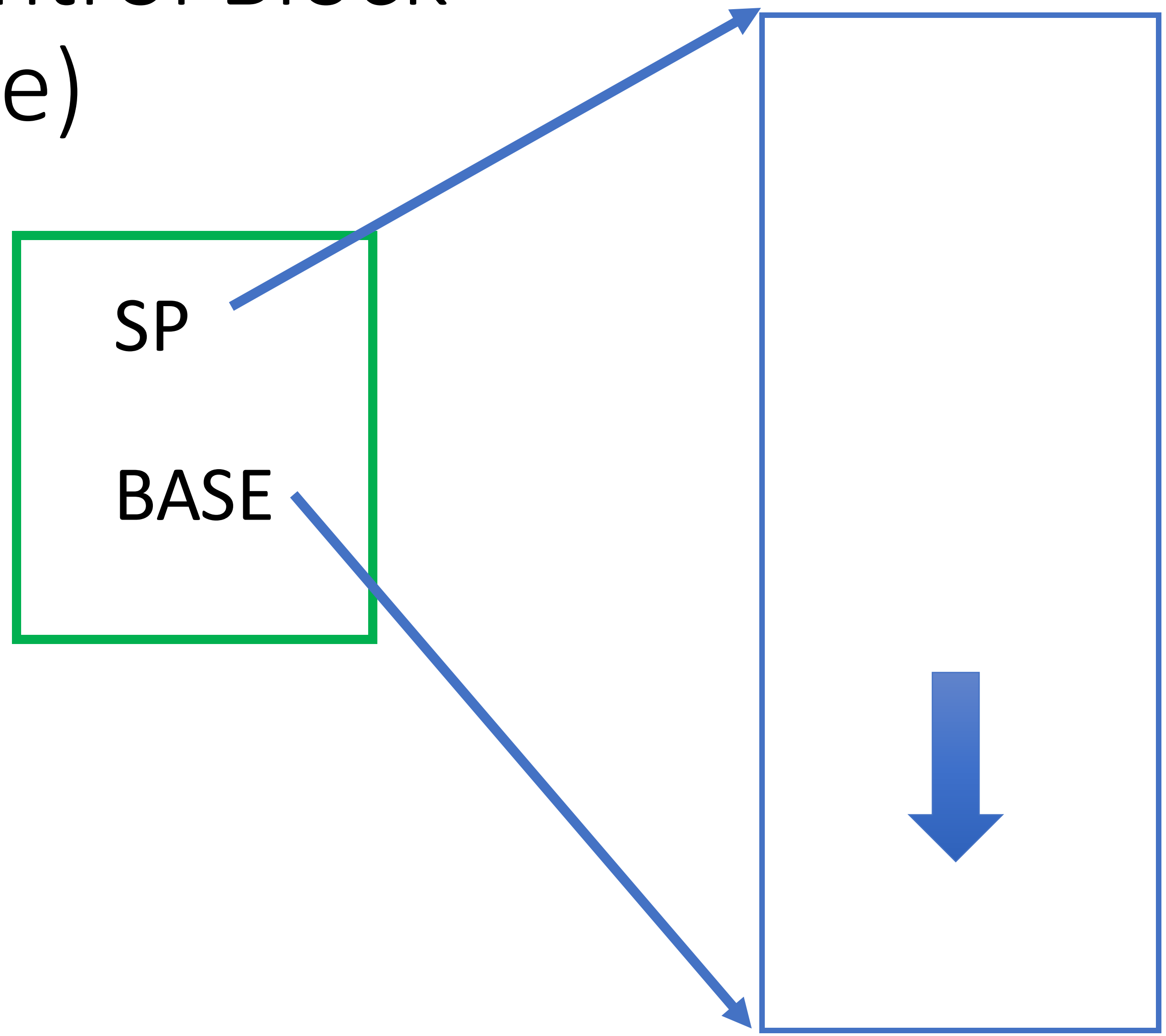
# Where to store the context of a thread?

- Convenient to push a thread’s registers onto the stack
  - but you can’t save the stack pointer on the stack…
- Keep the stack pointer in a *Thread Control Block*
  - one TCB per thread

# Thread Control Block



# Thread Control Block (initial state)



# Scheduling State of a Thread

- Running
  - currently running
- Runnable (aka Ready)
  - TCB on the run queue (aka ready queue)
- Terminated
  - TCB on the zombie queue

# thread_init()

- Initializes thread package
- Maintains global variables:
  - *run queue, zombie queue,* and *current thread*
- Initial run queue and zombie queues are empty
- 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
- 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
  - Check to see if there are any threads on the zombie queue

# thread_exit()

- See if the run queue is empty
  - if so, exit from the process using exit(0)
- Put TCB on zombie queue
- Get next TCB of the run queue
- **Switch contexts**
  - As before
- Next thread cleans up last thread!

# ctx_switch(&old_sp, new_sp)

```
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
```

USAGE:

```
struct tcb *current;
struct queue run_queue, zombie_queue;

void thread_yield(){
        struct tcb *old = current;
        run_queue.add(old);
        current = scheduler();
        if (current == old) return;
        ctx_switch(&old->sp, current->sp)
        while (zombie_queue is not empty) …
}
```

# Starting a *new* thread

```
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
   call    *%rdx
```

```
void ctx_entry(){
        (*current->func)();
        thread_exit();
        // this location cannot be reached
}

void thread_create( func ){
        struct tcb *old = current;
        runQueue.add(old);
        current = malloc(sizeof(struct tcb));
        current->func = func;
        current->stack  = malloc(…);
        ctx_start(&old->sp,  top of stack, ctx_entry)
        while (zombie_queue is not empty) …
}
```

# Come to front if you need a partner

# Synchronization Primitives

# Semaphores

- We’re not teaching general semaphores in CS4410 anymore
- A semaphore is a kind of counter:

```
struct sema;
void sema_init(struct sema *sema, unsigned int count);
void sema_dec(struct sema *sema);
void sema_inc(struct sema *sema);
bool sema_release(struct sema *sema);
```

# Semaphore interface

void sema_init(struct sema *sema, unsigned int count)

- Initialize the semaphore to the given counter

void sema_dec(struct sema *sema)

- Wait until sema > 0, then decrement semaphore

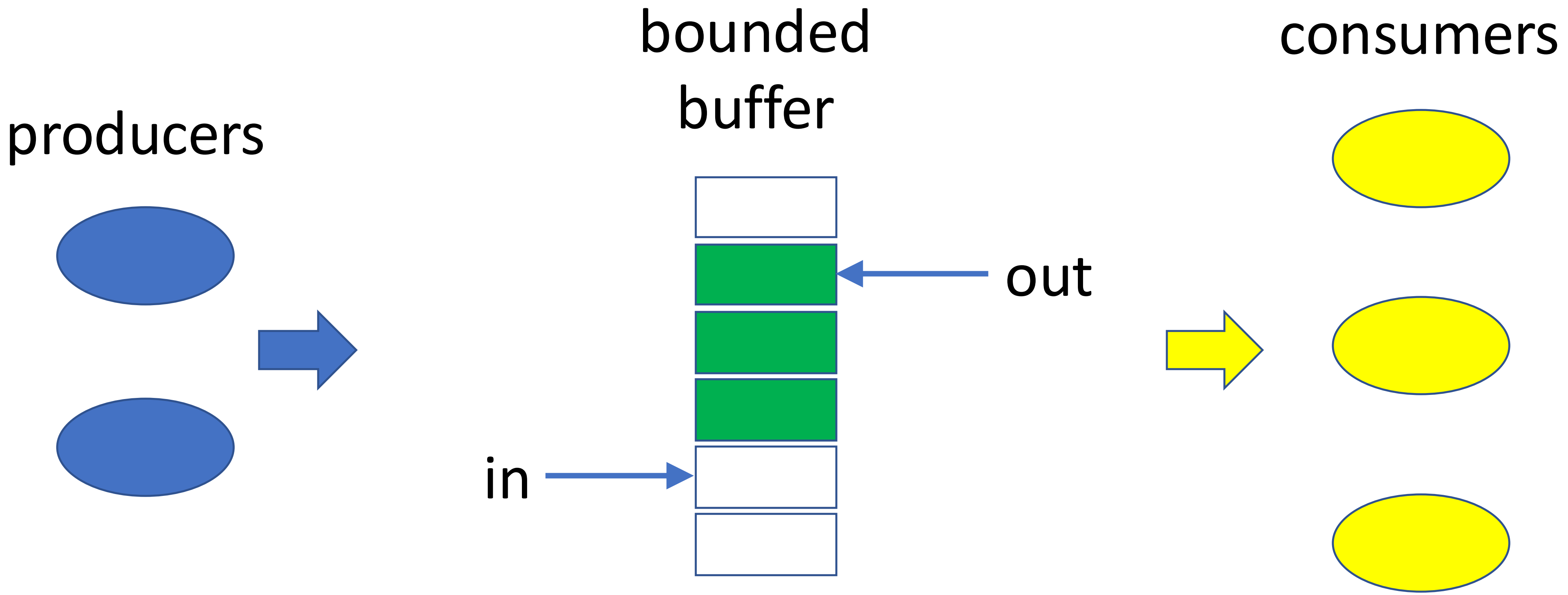
void sema_inc(struct sema *sema)

- Increment the semaphore

bool sema_release(struct sema *sema)

- Release the semaphore

# Example usage: Producer/Consumer




Producers block when buffer is full

Consumers block when buffer is empty

# Example usage: Producer/Consumer

```
#define NSLOTS    3

static struct sema s_empty, s_full, s_lock;
static unsigned int in, out;
static char *slots[NSLOTS];
```

```
int main(int argc, char **argv){
    thread_init();
    sema_init(&s_lock, 1);
    sema_init(&s_full, 0);
    sema_init(&s_empty, NSLOTS);

    thread_create(consumer, "consumer 1", 16 * 1024);
    producer("producer 1");
    thread_exit();
}
```

# Example usage: Producer/Consumer

```
static void producer(void *arg){
    for (;;) {
        // first make sure there's an empty slot.
        sema_dec(&s_empty);

        // now add an entry to the queue
        sema_dec(&s_lock);
        slots[in++] = arg;
        if (in == NSLOTS) in = 0;
        sema_inc(&s_lock);

        // finally, signal consumers
        sema_inc(&s_full);
    }
}
```

# Example usage: Producer/Consumer

```
static void consumer(void *arg){
    unsigned int i;

    for (i = 0; i < 5; i++) {
        // first make sure there's something in the buffer
        sema_dec(&s_full);

        // now grab an entry to the queue
        sema_dec(&s_lock);
        void *x = slots[out++];
        printf("%s: got '%s'\n", arg, x);
        if (out == NSLOTS) out = 0;
        sema_inc(&s_lock);

        // finally, signal producers
        sema_inc(&s_empty);
    }
}
```

# Semaphore implementation

- Associate a counter and a queue with each semaphore
- If thread tries to decrement a semaphore with a zero counter, put its TCB on the semaphore queue
    - otherwise decrement the counter
- If thread increments a semaphore with a non-empty queue, don't increment the counter but move one TCB from the semaphore's queue to the ready queue
    - otherwise increment the counter

# On Testing

# Tip 1: use assertions in your implementation (and not in your test code)

- Pepper your code with assertions before testing
  - think carefully about invariants
  - check invariants as often as possible
  - write code to check invariants

# Quick aside on using assertions

- assert( *P* ) --- executable *comment*
- Important: *P* should have no side effects
  - so, don't do assert(sema_release(s))
- assert statements should be no-ops and can be turned off
- use assert statements to check correctness, not to detect failures
  - so, don't do p = malloc(); assert(p != NULL)
- split conjunctions
  - so, don't do assert(P && Q) but do assert(P); assert(Q)

# Tip 2: don’t ignore warnings

- Compile with –g –Wall
  - e.g., cc –g –Wall x.c
- Do *not* submit code with outstanding warnings
- Do *not* get rid of warnings by hasty casting
  - Be very careful and only cast if you know exactly what you’re doing

# Tip 3: run small tests

- Don’t run very large tests (10s of operations or more)
    - you are unlikely to find bugs that you can’t find with small tests
    - it’s hard to figure out what went wrong
    - tests may take a long time for no good reason

# Tip 4: use valgrind

- Will immediately notify you if
  - you are using uninitialized memory (e.g., from malloc())
  - you are accessing illegal memory
  - you are leaking memory
- It will give you lots of information about how it happened
- Easiest to install under Linux, so use a virtual machine or log into CSUGlab Linux machines

# Tip 5: only check things that are specified

- Carefully read the spec and design tests for each specified case
- Do not check things that are not specified
  - sema_dec(NULL) has unspecified behavior---don’t test it

# Tip 6: think carefully about corner cases

- be systematic

# Tip 7: test your test program

- don’t just run it against your own implementation
- take your implementation and break it in various ways
- see if your test program notices