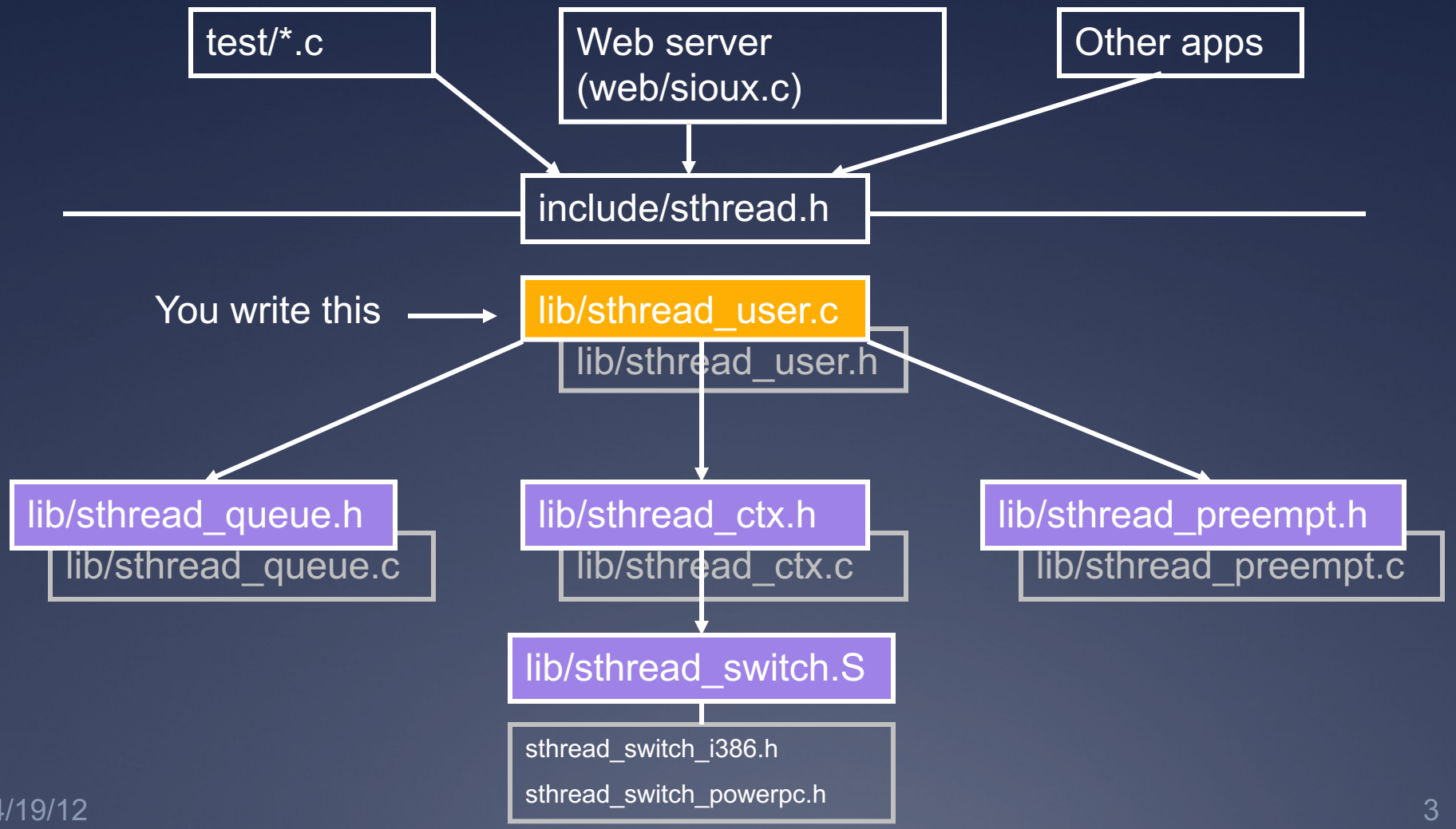
CSE 451: Operating Systems

Simplethreads

Simplethreads

- We give you:
  - Skeleton functions for thread interface
  - Machine-specific code (x86, i386)
    - Support for creating new stacks
    - Support for saving regs/switching stacks
  - A queue data structure (why?)
  - Very simple test programs
    - You should write more, and include them in the turnin

Simplethreads code structure



Pthreads

- Pthreads (POSIX threads) is a preemptive, kernel-level thread library
- Simplethreads is similar to Pthreads
- Project 2: compare your implementation against Pthreads
  - ./configure --with-pthreads

Thread operations

- What functions do we need for a userspace thread library?

Simplethreads API

```
void sthread_init()
```

- Initialize the whole system

```
sthread_t sthread_create(func start_func,
   void *arg)
```

- Create a new thread and make it runnable

```
void sthread_yield()
```

- Give up the CPU

```
void sthread_exit(void *ret)
```

- Exit current thread

```
void* sthread_join(sthread_t t)
```

- Wait for specified thread to exit

Simplethreads internals

- Structure of the TCB:

```
struct _sthread {
    sthread_ctx_t *saved_ctx;
    /**
     * Add your fields to the thread
     * data structure here.
     */
};
```

Sample multithreaded program

* (this slide and next – see test-create.c)

```
void *thread_start(void *arg) {
   if (arg) {
      printf("in thread_start, arg = %p\n",
         arg);
   }
   return 0;
}

...
```

Sample multithreaded program

```
int main(int argc, char *argv[]) {
   sthread_init();
   for(i = 0; i < 3; i++) {
      if (sthread_create(thread_start,
                         (void *)&i) == NULL) {
         printf("sthread_create failed\n");
         exit(1);
      }
   }
   // needs to be called multiple times
   sthread_yield();
   printf("back in main\n");
   return 0;
}
```

Managing contexts

- (Provided for you in project 2)
- Thread *context* = thread stack + stack pointer

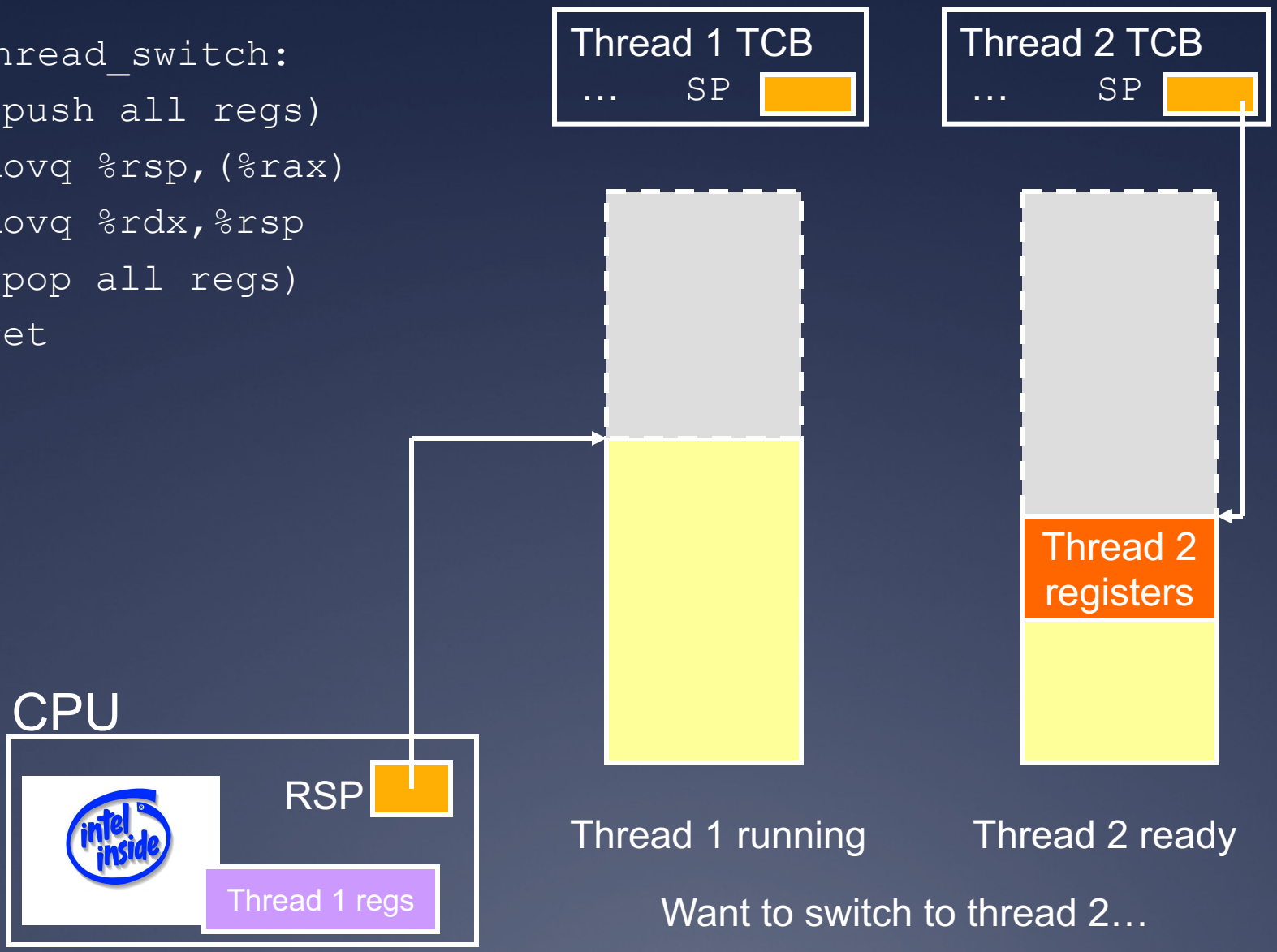
`sthread_new_ctx(func_to_run)`
  - creates a new thread context that can be switched to

`sthread_free_ctx(some_old_ctx)`
  - Deletes the supplied context

`sthread_switch(oldctx, newctx)`
  - Puts current context into oldctx
  - Takes newctx and makes it current

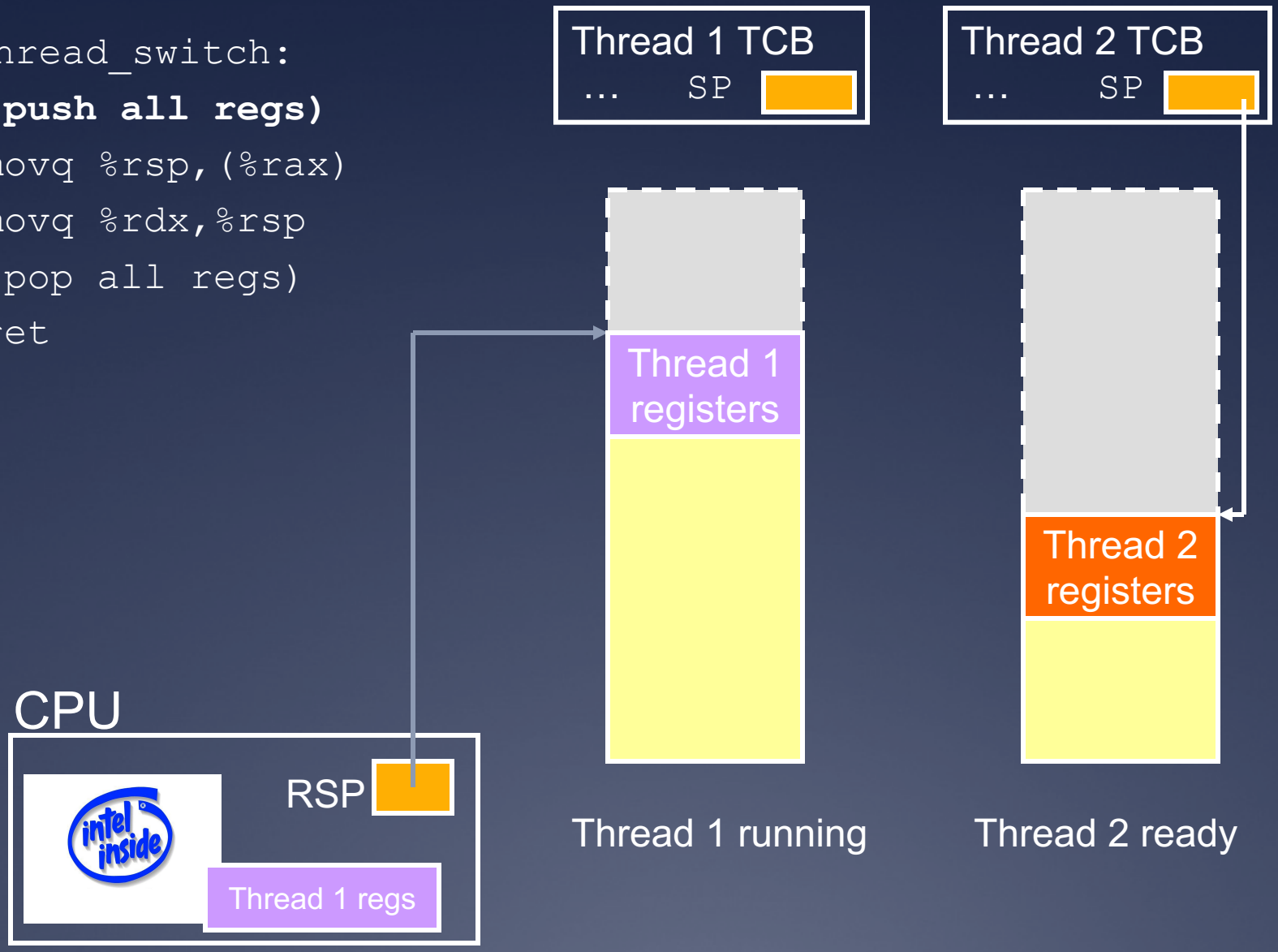
How sthread_switch works

```
Xsthread_switch:
   (push all regs)
   movq %rsp,(%rax)
   movq %rdx,%rsp
   (pop all regs)
   ret
```



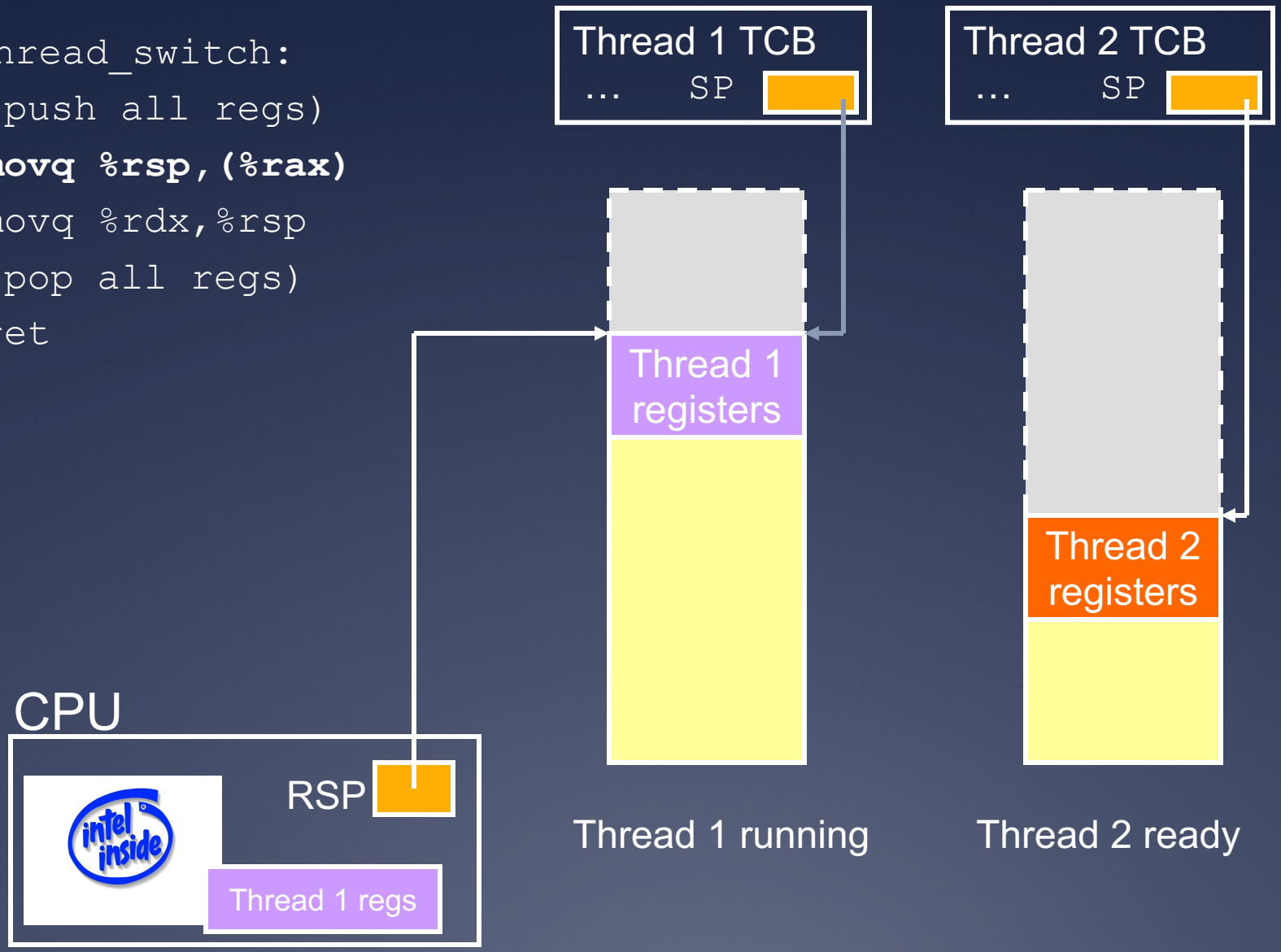
Push old context

```
Xsthread_switch:
    (push all regs)
    movq %rsp,(%rax)
    movq %rdx,%rsp
    (pop all regs)
    ret
```



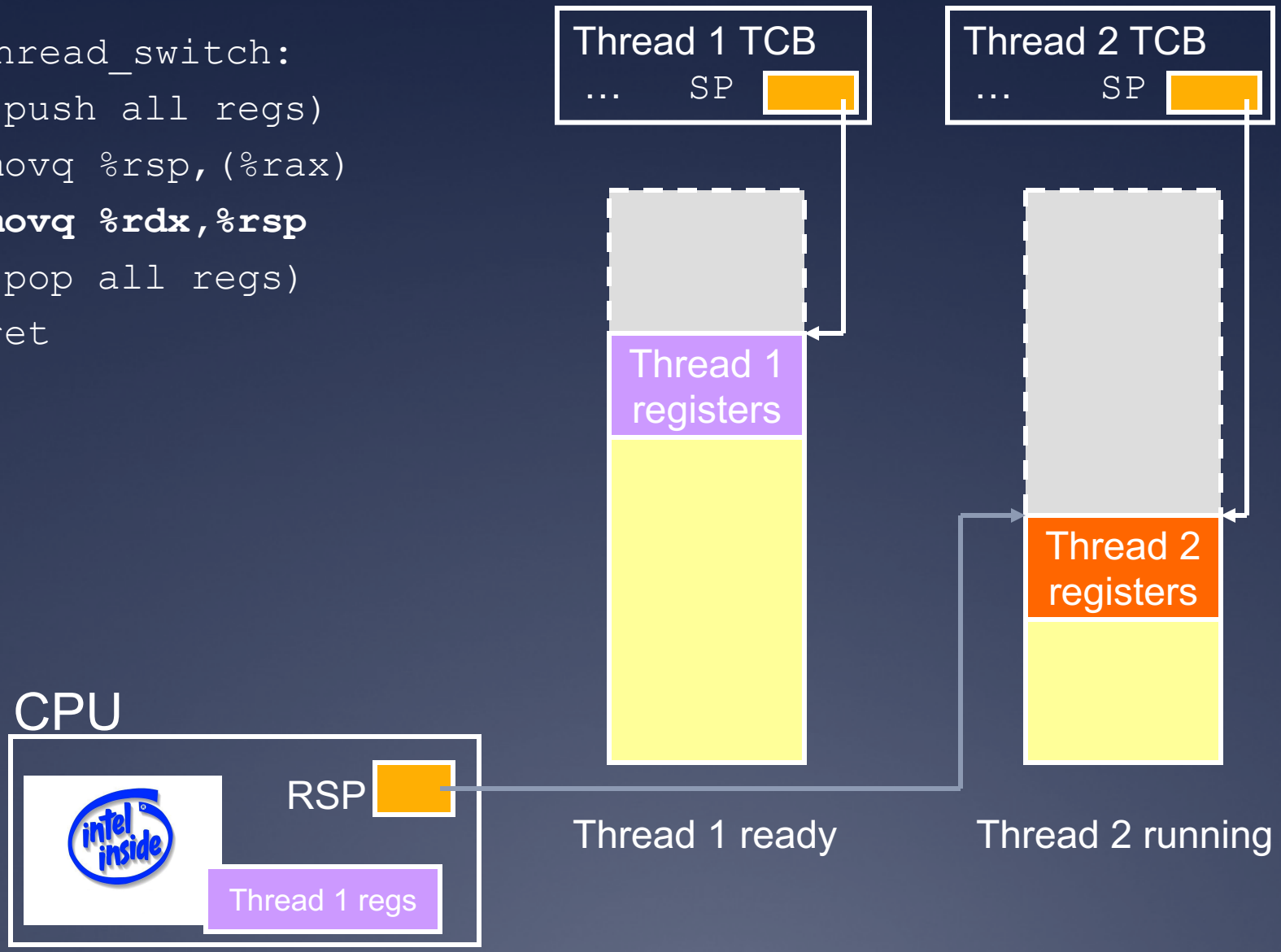
Save old stack pointer

```
Xsthread_switch:
   (push all regs)
   movq %rsp,(%rax)
   movq %rdx,%rsp
   (pop all regs)
   ret
```



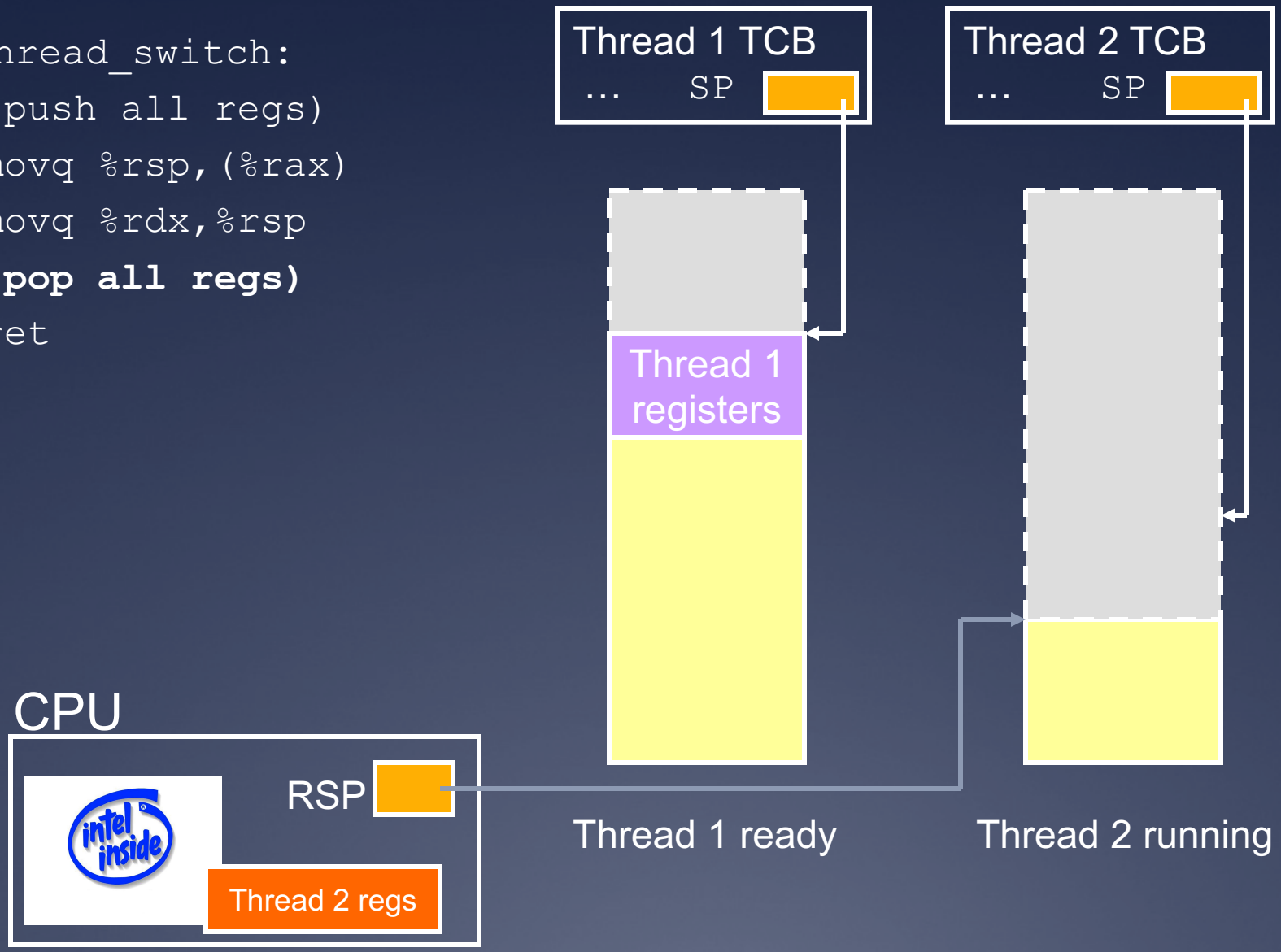
Change stack pointers

```
Xsthread_switch:
    (push all regs)
    movq %rsp,(%rax)
    movq %rdx,%rsp
    (pop all regs)
    ret
```



Pop off new context

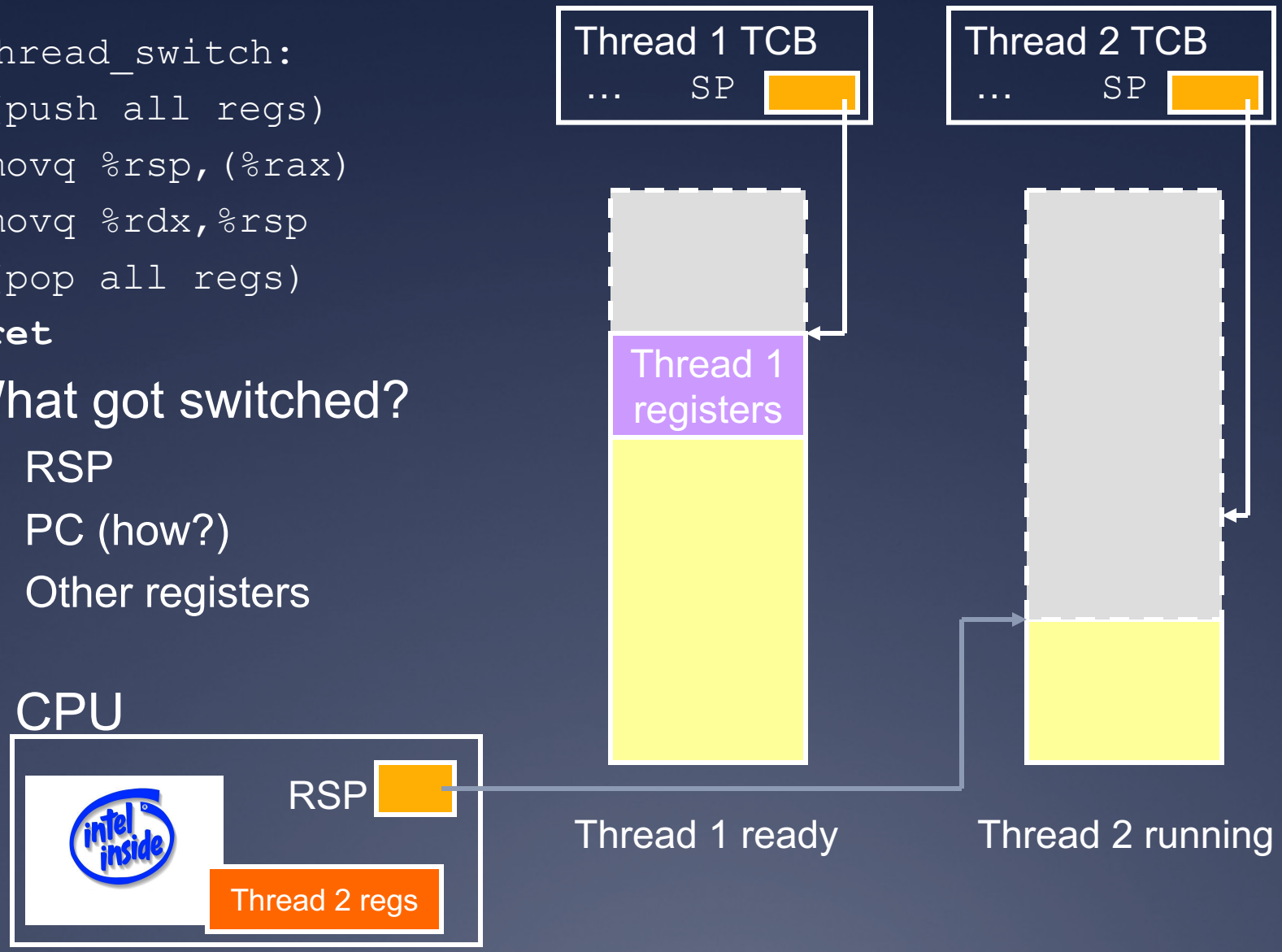
```
Xsthread_switch:
    (push all regs)
    movq %rsp,(%rax)
    movq %rdx,%rsp
    (pop all regs)
    ret
```



Done; return

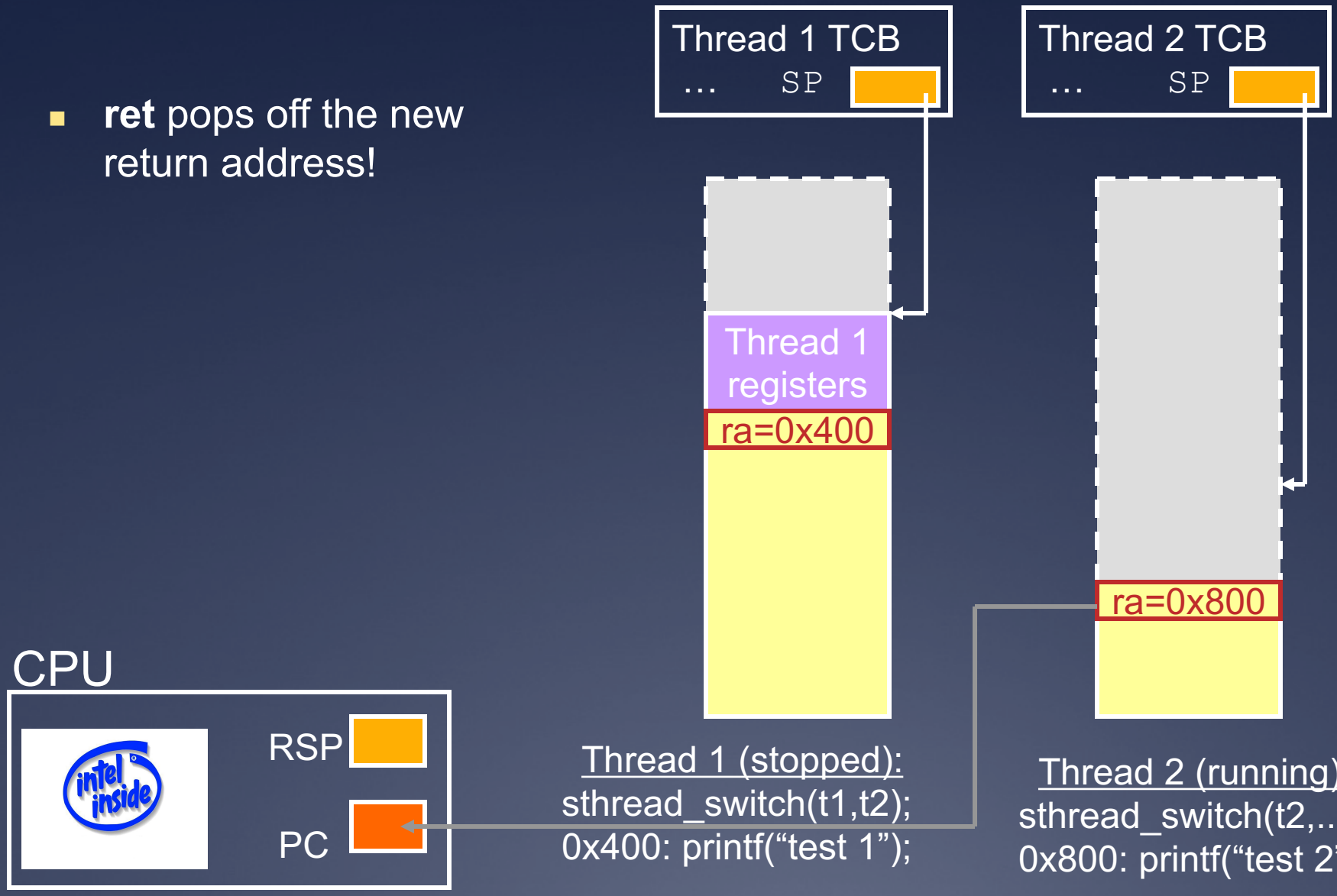
```
Xsthread_switch:
    (push all regs)
    movq %rsp,(%rax)
    movq %rdx,%rsp
    (pop all regs)
    ret
```

- What got switched?
  - RSP
  - PC (how?)
  - Other registers



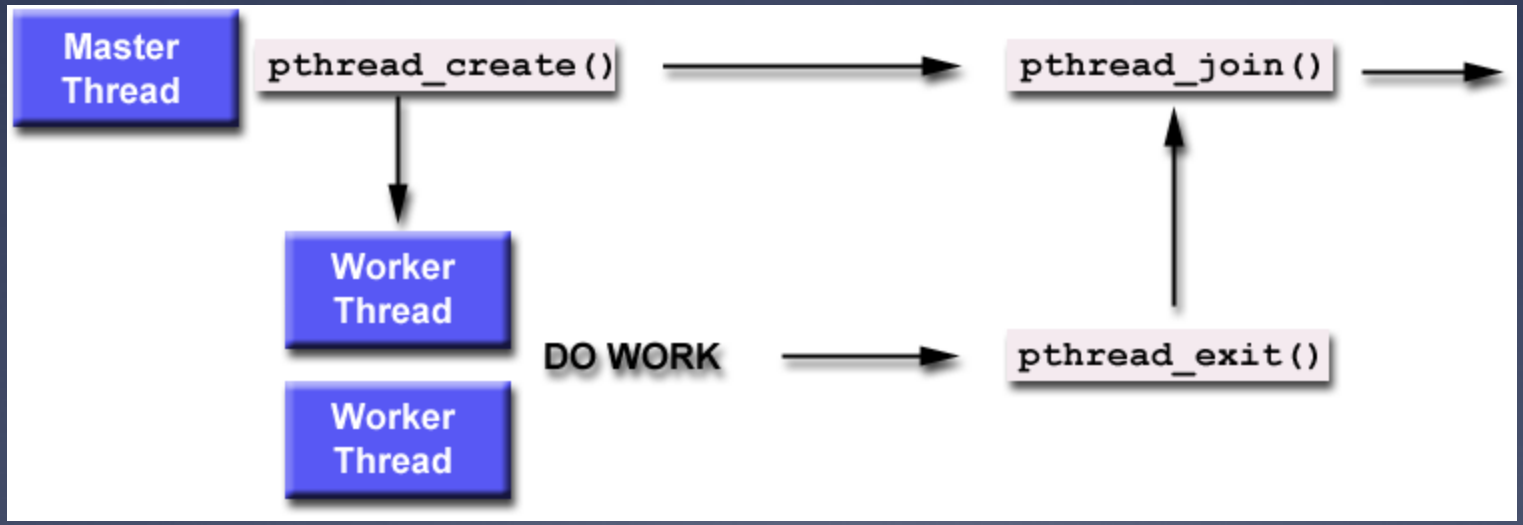
Adjusting the PC

- **ret** pops off the new return address!



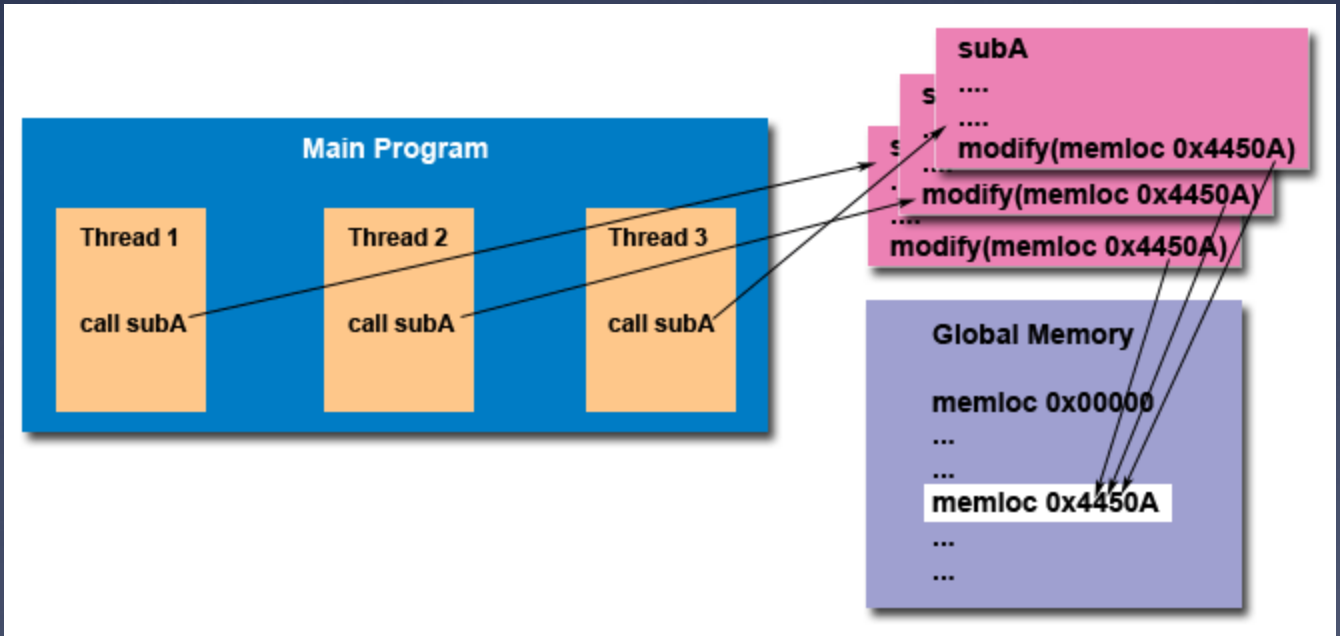
Thread joining

- With Pthreads (and Sthreads):
  - Master thread calls join on worker thread
  - Join blocks until worker thread exits.
  - Join returns the return value of the worker thread.



The need for synchronization

- Thread safety:
  - An application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions



Synchronization primitives: mutexes

```
sthread_mutex_t sthread_mutex_init()
void sthread_mutex_free(sthread_mutex_t lock)

void sthread_mutex_lock(sthread_mutex_t lock)
```

* When returns, thread is guaranteed to acquire lock

```
void sthread_mutex_unlock(
   sthread_mutex_t lock)
```

Synchronization primitives: condition variables

```
sthread_cond_t sthread_cond_init()
void sthread_cond_free(sthread_cond_t cond)

void sthread_cond_signal(sthread_cond_t cond)
```

- Wake-up one waiting thread, if any

```
void sthread_cond_broadcast(
   sthread_cond_t cond)
```

- Wake-up all waiting threads, if any

```
void sthread_cond_wait(sthread_cond_t cond,
   sthread_mutex_t lock)
```

- Wait for given condition variable
- Returning thread is guaranteed to hold the lock

Things to think about

- How do you create a thread?
  - How do you pass arguments to the thread's start function?
    - Function pointer passed to sthread_new_ctx() doesn't take any arguments
- How do you deal with the initial (main) thread?
- How do you block a thread?

Things to think about

- When and how do you reclaim resources for a terminated thread?
  - Can a thread free its stack itself?
- Where does sthread_switch return?
- Who and when should call sthread_switch?
- What should be in struct _sthread_mutex, struct _sthread_cond?

sthread_preempt.h

```
/* Start preemption - func will be called
 * every period microseconds
 */
void sthread_preemption_init
   (sthread_ctx_start_func_t func,
    int period);

/* Turns interrupts on (LOW) or off (HIGH)
 * Returns the last state of the
 * interrupts
 */
int splx(int splval);
```

sthread_preempt.h

```
/* atomic_test_and_set - using the native
 * compare and exchange on the Intel x86.
 *
 * Example usage:
 *    lock_t lock;
 *    while(atomic_test_and_set(&lock))
 *      {} // spin
 *    _critical section_
 *    atomic_clear(&lock);
 */
int atomic_test_and_set(lock_t *l);
void atomic_clear(lock_t *l);
```

What you need to do

- Add a call to sthread_preemption_init() as the last line in your sthread_user_init() function
  - sthread_preemption_init() takes a pointer to a function that will be called on each timer interrupt
    - This function should cause thread scheduler to switch to a different thread!

What you need to do

- Add synchronization to *critical sections* in thread management routines
  - Think: what would happen if the code was interrupted at this point?
    - Would it resume later with no problems?
    - Could the interrupting code mess with any variables that this code is currently using?
  - Don't have to worry about simplethreads code that you didn't write (i.e. sthread_switch): already done for you

What you need to do

- Before doing a context switch, interrupts should be disabled to avoid preemption. How can they be reenabled after the switch?
  - Hint: Think of the possible execution paths

Interrupt disabling

Non-thread-safe

```
/* returns next thread
 * on the ready queue */
sthread_t
sthread_user_next() {
  sthread_t next;
  next = sthread_dequeue
(ready_q);
  if (next == NULL)
    exit(0);
  return next;
}
```

Thread-safe

```
sthread_t
sthread_user_next() {
  sthread_t next;
  int old = splx(HIGH);
  next = sthread_dequeue
                (ready_q);
  splx(old);
  if (next == NULL)
    exit(0);
  return next;
}
```

Interrupt disabling

- Why do we call `splx(old)` after dequeuing instead of just `splx(LOW)`?

Thread-safe

```
sthread_t
sthread_user_next() {
  sthread_t next;
  int old = splx(HIGH);
  next = sthread_dequeue
                (ready_q);
  splx(old);
  if (next == NULL)
    exit(0);
  return next;
}
```

Atomic locking

- So what is `atomic_test_and_set()` for?
  - Primarily to implement higher-level synchronization primitives (mutexes, CVs)
- One way to think about preemption-safe thread library:
  - Disable/enable interrupts in “library” context
  - Use atomic locking in “application” context

Race conditions and testing

- How can you test your preemption code?
- How can you know that you've found all of the critical sections?

Part 5: report

- Covers *all* parts of project 2
- Discuss your design decisions. In detail. PLEASE!