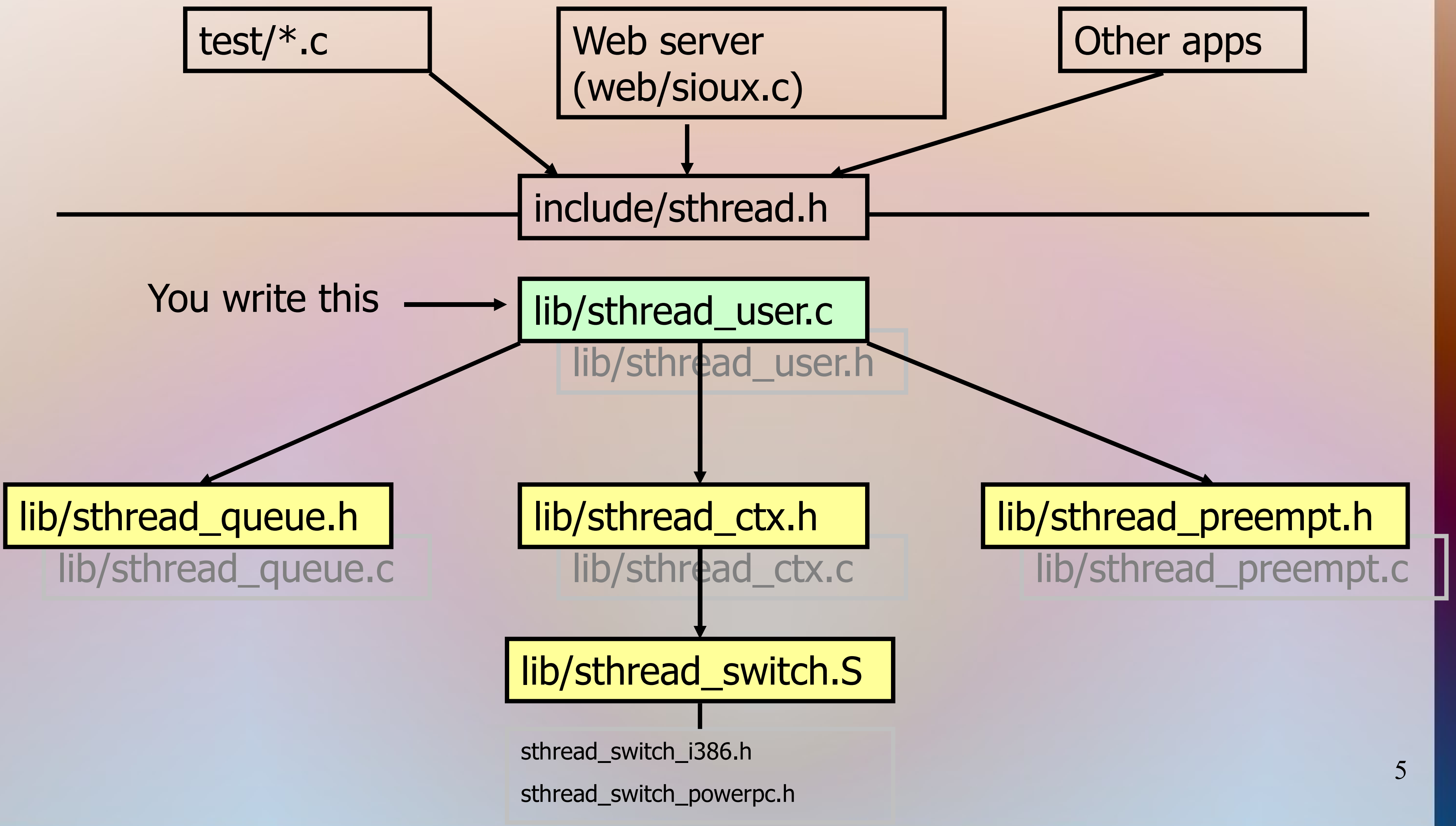
Project 2

- You have to:
  - Part a:
    - Implement a user thread library
    - Implement synchronization primitives
    - Solve a synchronization problem
  - Part b:
    - Add Preemption
    - Implement a multithreaded web server
    - Get some results and write a (small) report
- Part a and b due separately
  - Part a due Friday May 8th, 11:59pm
  - Part b due Friday May 22nd, 11:59pm

Simplethreads

- We give you:
  - Skeleton functions for thread interface
  - Machine-specific code
    - Support for creating new stacks
    - Support for saving regs/switching stacks
  - A generic queue
    - When do you need one?
  - Very simple test programs
    - You should write more, and include them in the turnin
  - Singlethreaded web server

Simplethreads Code Structure



Thread Operations

- What functions do we need?
- What should the TCB look like?

Thread Operations

- `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

- Structure of the TCB:

```
struct _thread {
    sthread_ctx_t *saved_ctx;
    .........


}
```

Sample multithreaded program

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

   sthread_init();
   for(i=0; i<3; i++)
      if (sthread_create(thread_start, (void*)i) == NULL) {
          printf("sthread_create failed\n");
          exit(1);
      }

   sthread_yield();
   printf("back in main\n");
   return 0;
}

void *thread_start(void *arg) {
  printf("In thread_start, arg = %d\n", (int)arg);
  return 0;
}
```

- **Output? (assume no preemption)**

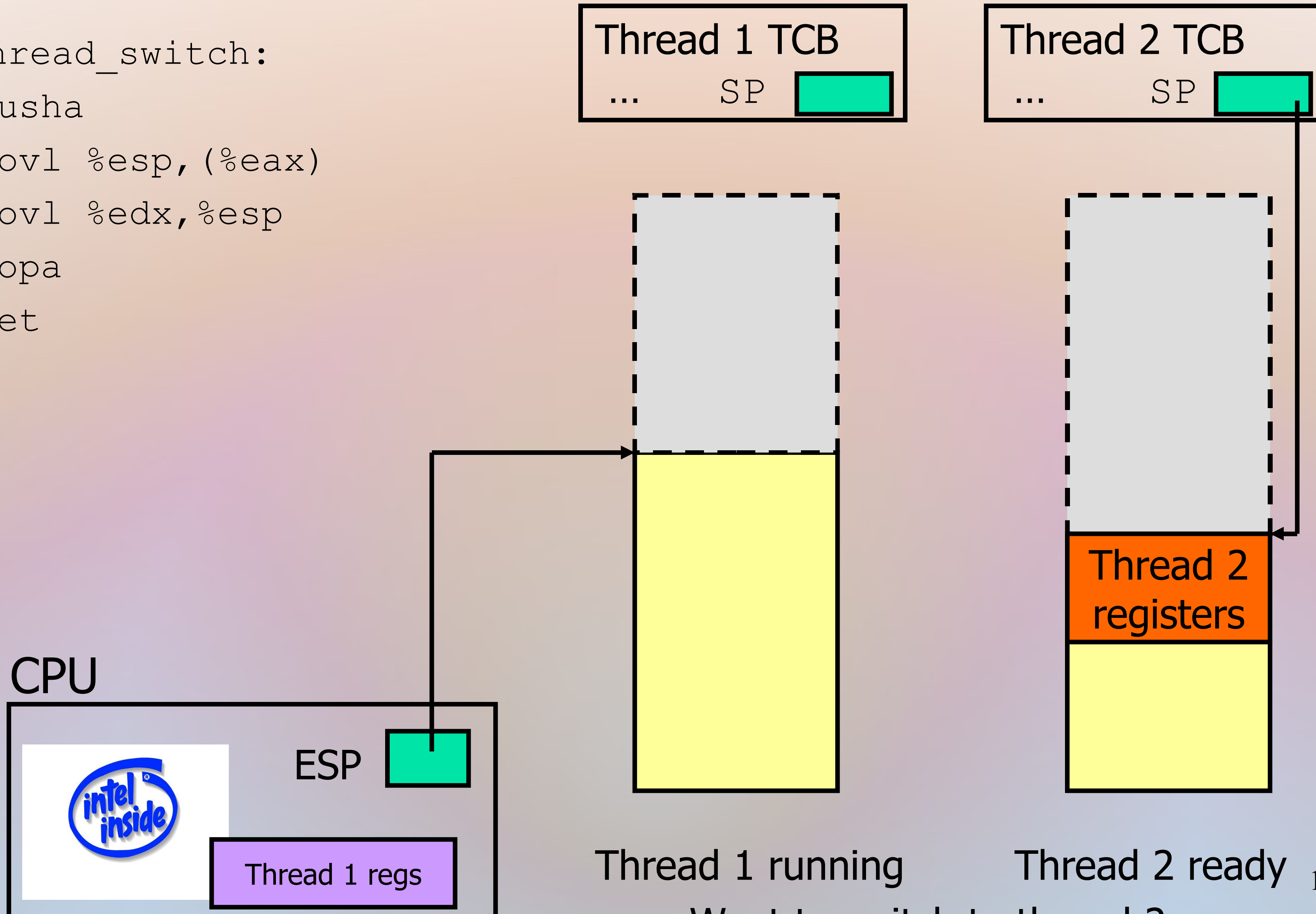
Managing Contexts (given)

- 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:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



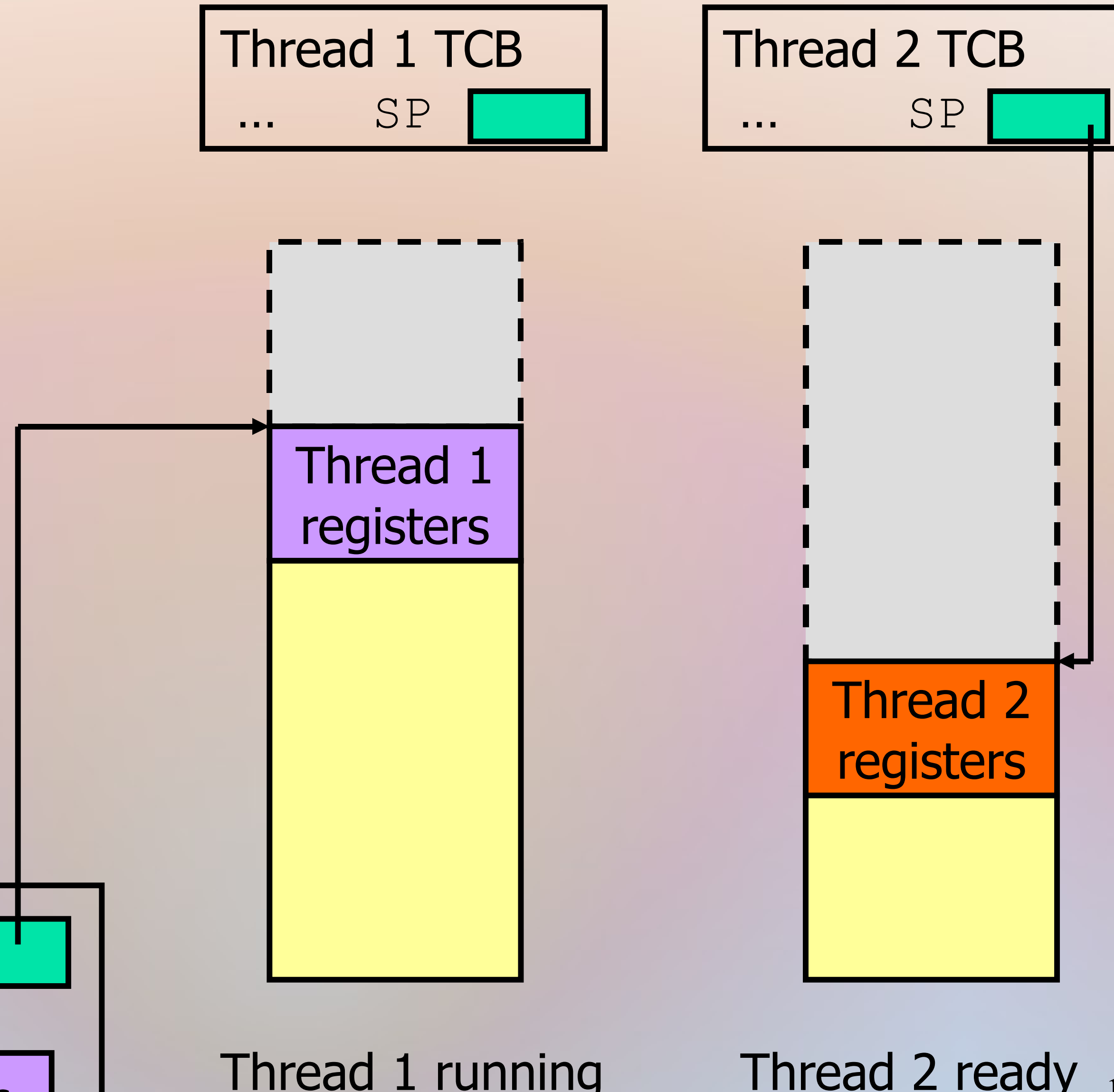
Thread 1 running

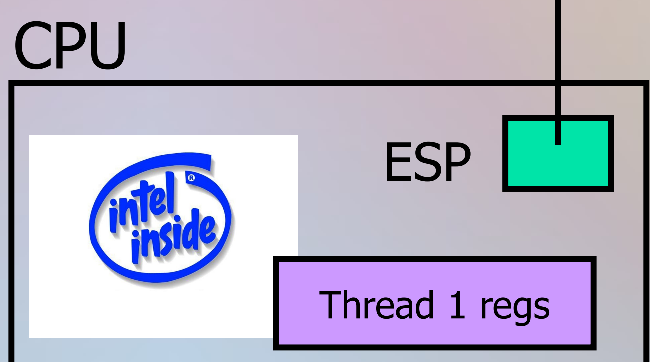
Thread 2 ready

Want to switch to thread 2…

Push old context

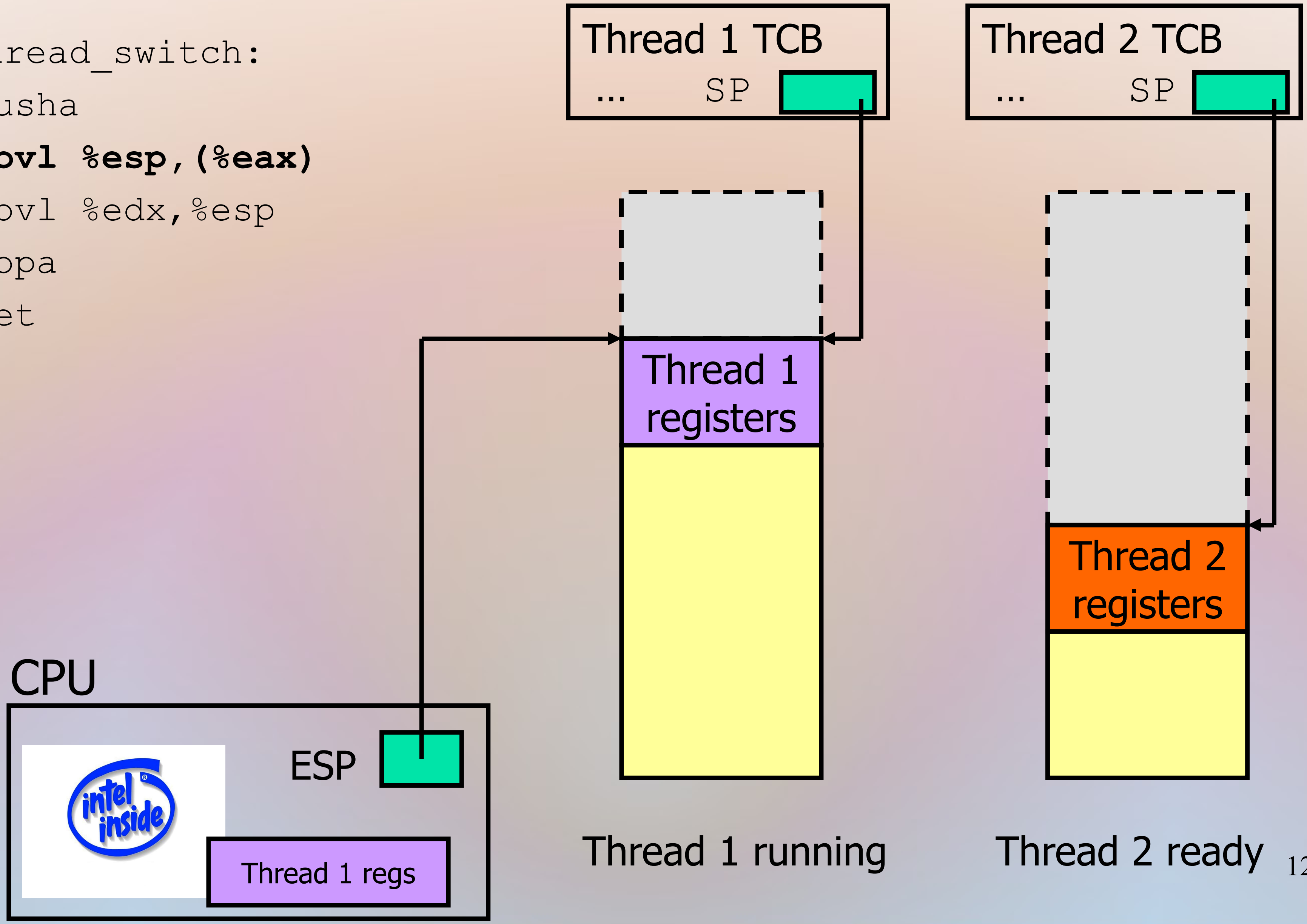
```
Xsthread_switch:
    pusha
    movl %esp,(%eax)
    movl %edx,%esp
    popa
    ret
```





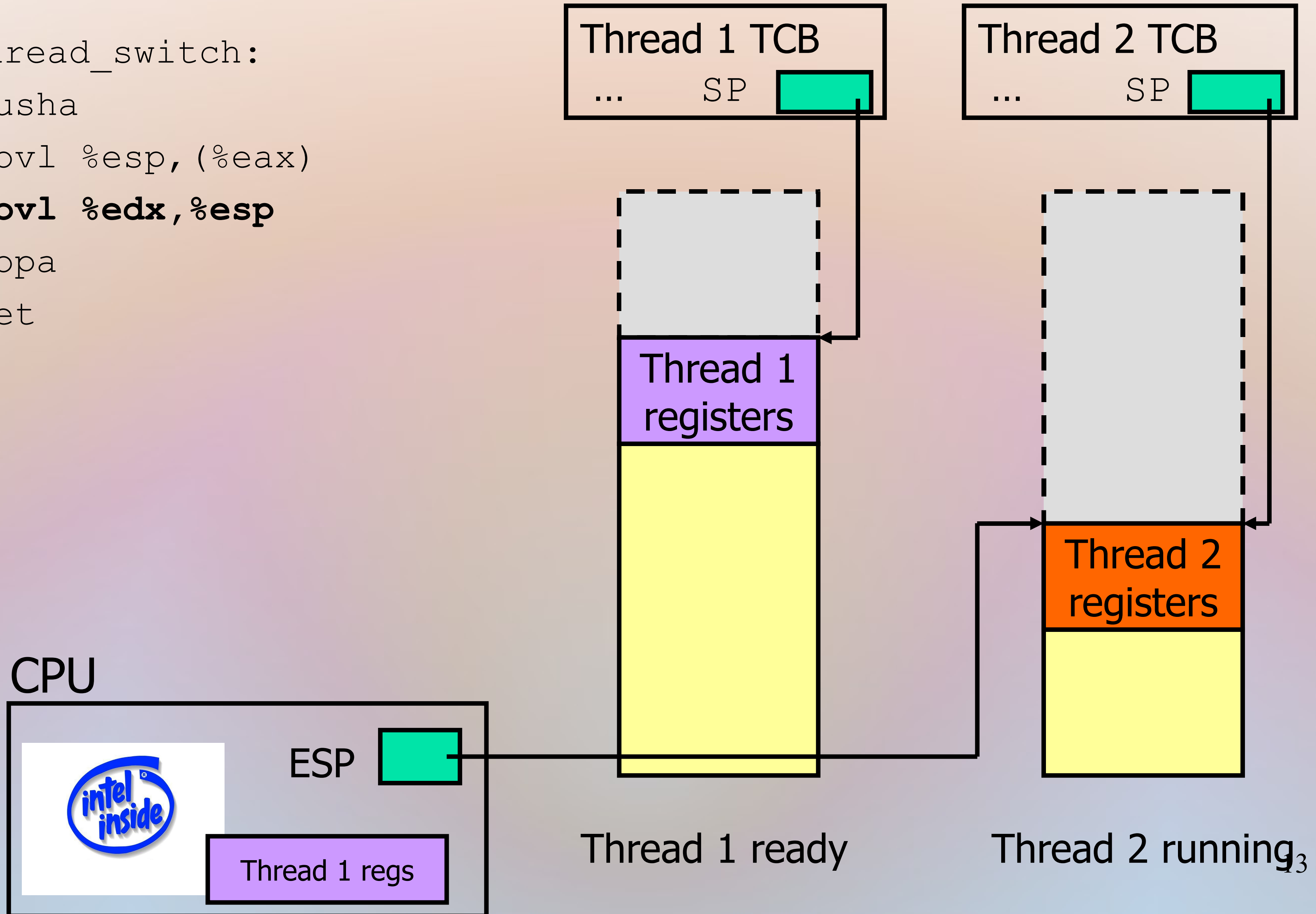
Save old stack pointer

```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



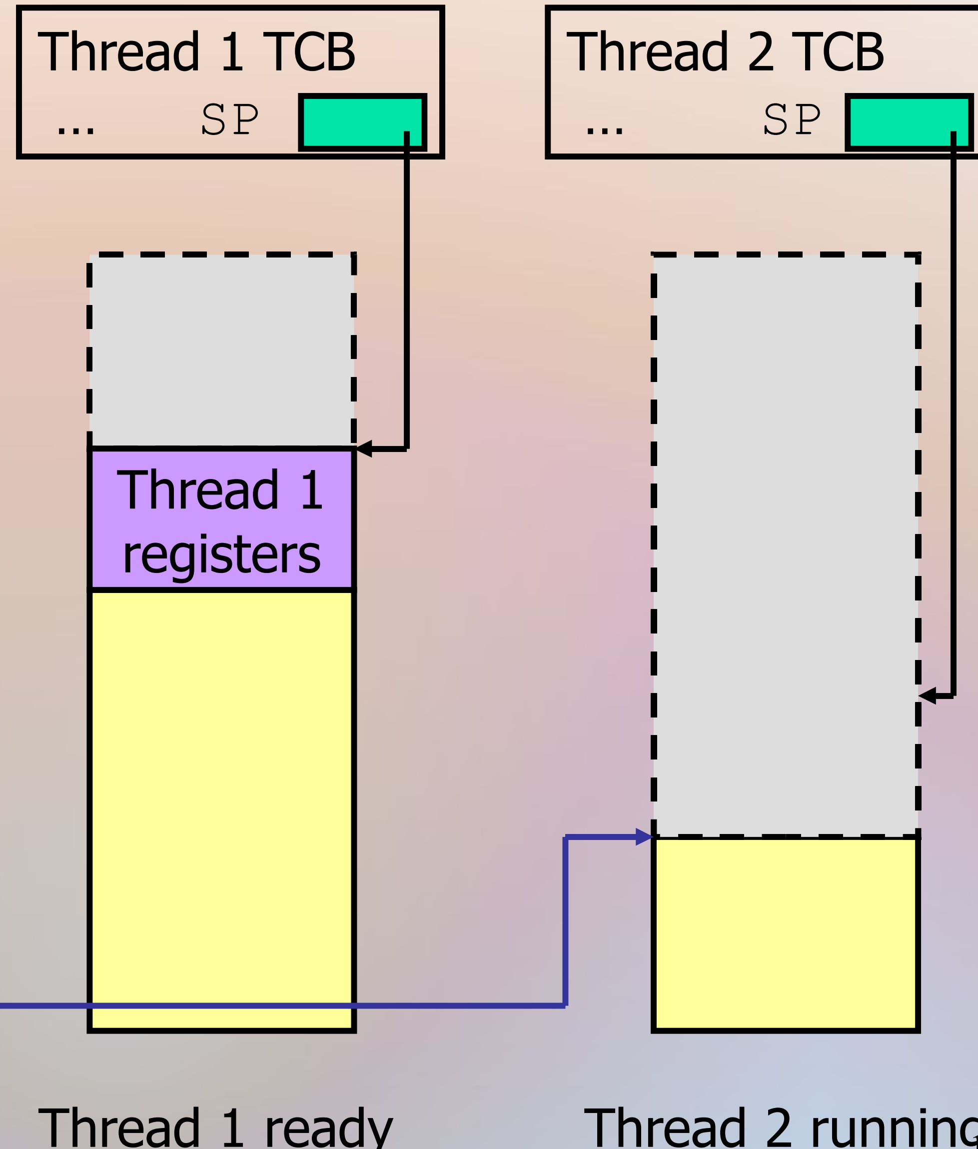
Change stack pointers

```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



Pop off new context

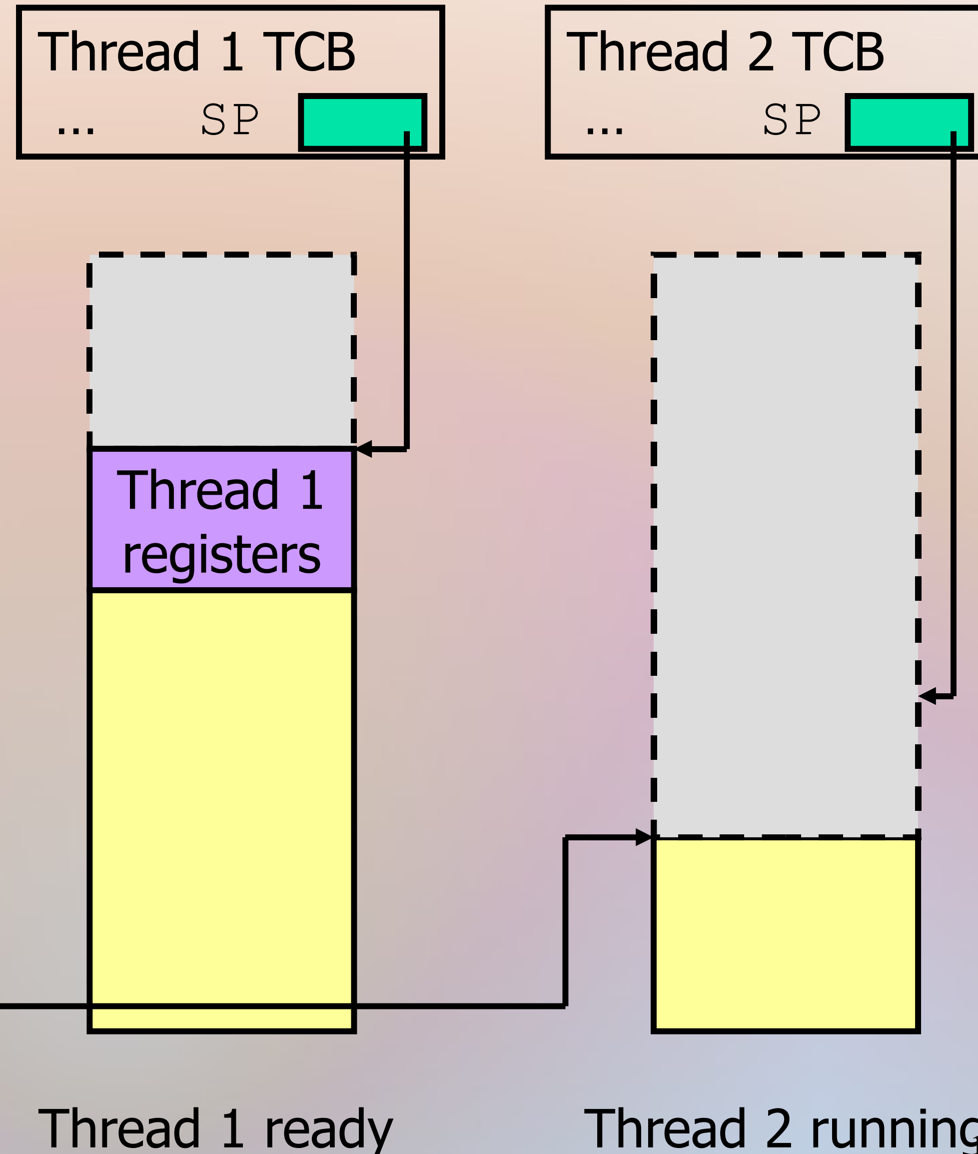
```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



Done; return

```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```

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

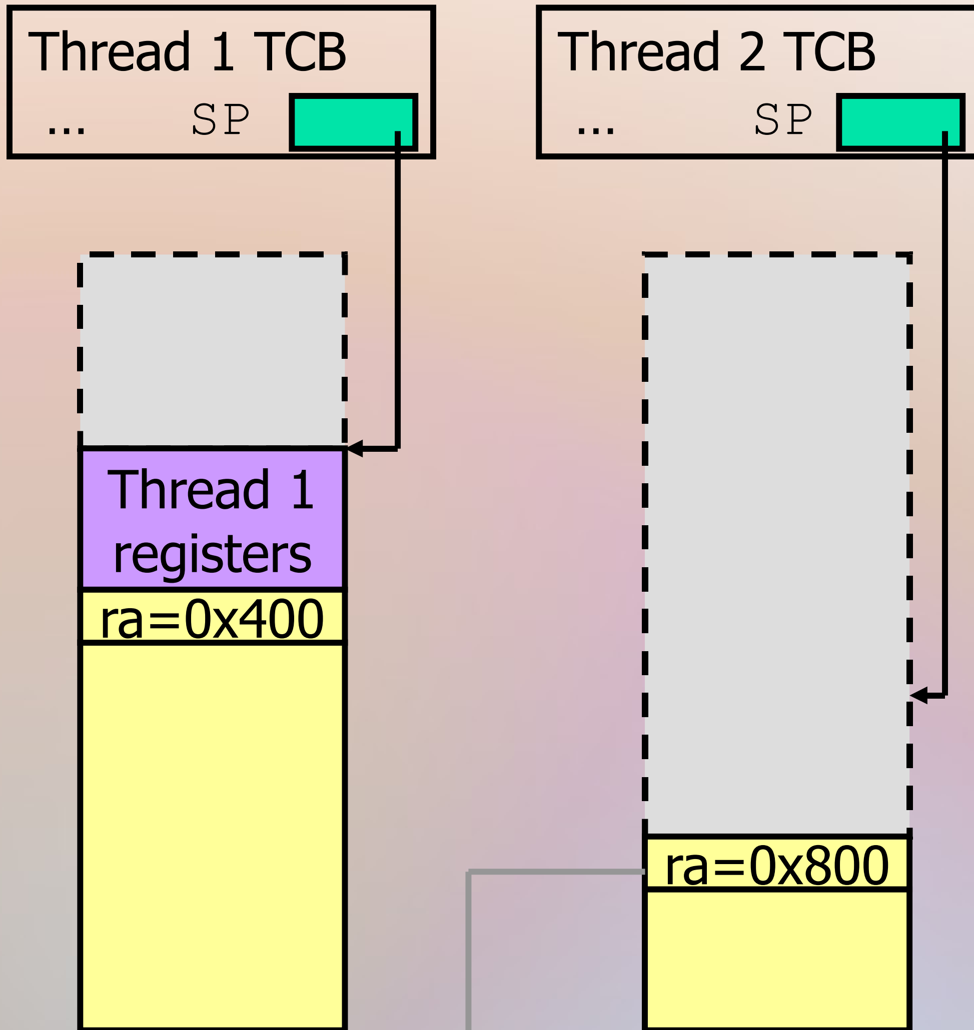


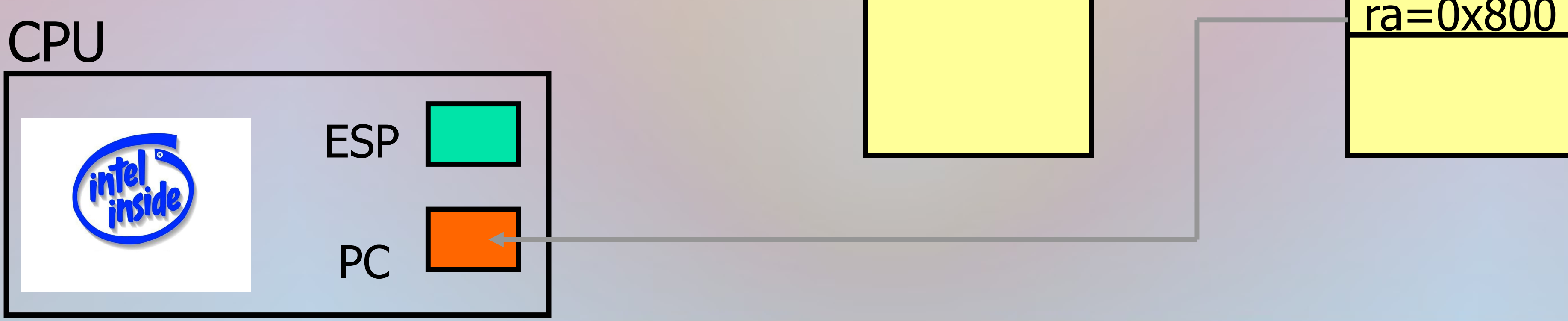
Adjusting the PC

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

Thread 1 (stopped):
switch(t1,t2);
0x400: printf("test 1");

Thread 2 running:
switch(t2,...);
0x800: printf("test 2");





Synchronization primitives: Mutex

- sthread_mutex_t sthread_mutex_init()
- void sthread_mutex_free(sthread_mutex_t lock)
- void sthread_mutex_lock(sthread_mutex_t lock)
  - Returned thread is guaranteed to acquire lock
- void sthread_mutex_unlock(sthread_mutex_t lock)
  - Release lock
- See sthread.h

Synch 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

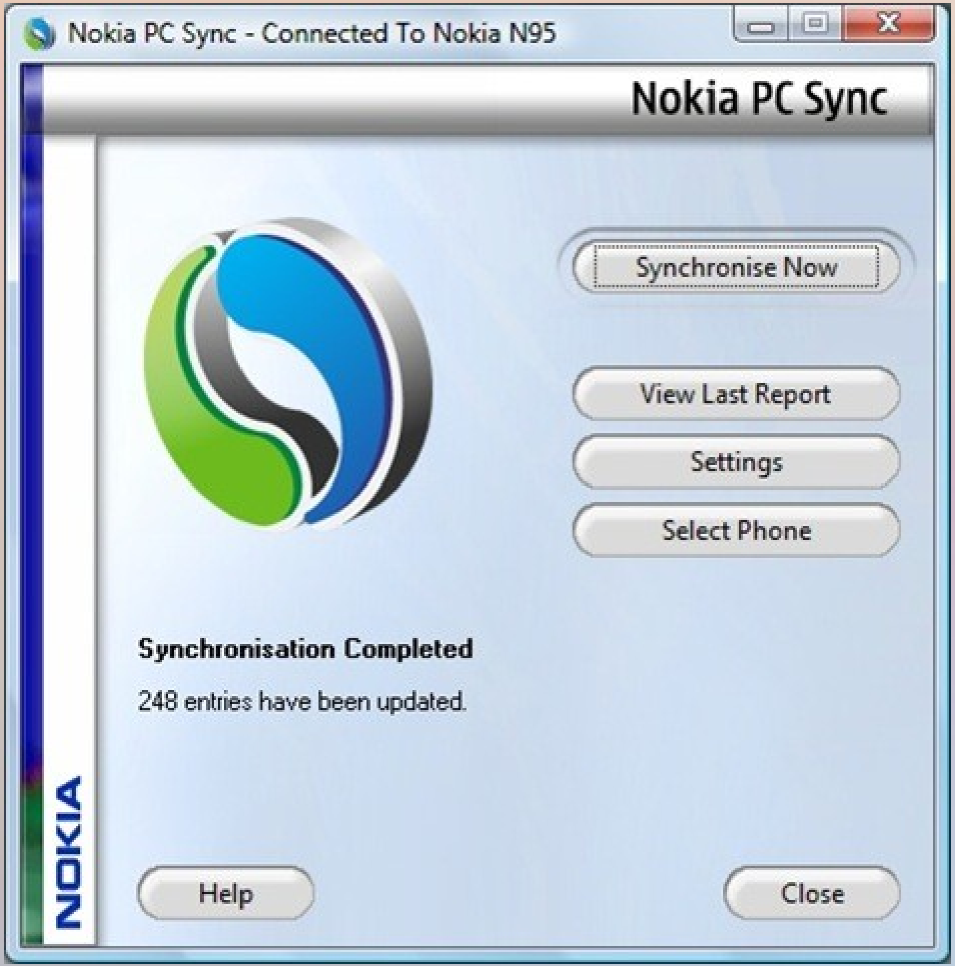
- How do you create a thread?
  - How do you pass arguments to the thread's start function?
  - (sthread_new_ctx() doesn't call function w/ arguments)
- How do you deal with the initial (main) thread?
- When/how do you free resources for a terminated thread?
  - Can a thread free its stack itself?
- Where does sthread_switch return?
- Who and when should call sthread_switch?
- How do you block a thread?
- What should be in struct _sthread_mutex|cond?

Sthread is similar to pthread

- Pthread (POSIX threads) is a preemptive, kernel-level thread library
- You can compare your implementation against pthreads
  - ./configure --with-pthreads

Synchronization primitives

What is synchronization?



Synchronization

High-level
- Monitors
- Java synchronized method

OS-level support
- Special variables – mutex, futex, semaphor, condition var
- Message passing primitives

Low-level support
- Disable/enable interrupts
- Atomic instructions (test_and_set)

Disabling/Enabling Interrupts

```
Thread A:
      disable_irq()
      critical_section()
      enable_irq()
```

```
Thread B:
      disable_irq()
      critical_section()
      enable_irq()
```

- Prevents context-switches during execution of critical sections
- Sometimes necessary
  - E.g. to prevent further interrupts during interrupt handling
- Many problems

Disabling/Enabling Interrupts

```
Thread A:
      disable_irq()
      critical_section()
      enable_irq()
```

```
Thread B:
      disable_irq()
      critical_section()
      enable_irq()
```

- Prevents context-switches during execution of critical sections
- Sometimes necessary
  - E.g. to prevent further interrupts during interrupt handling
- Many problems
  - E.g., an interrupt may be shared
  - How does it work on multi-processors?

Hardware support

- Atomic instructions:
  - test_and_set
  - Compare-exchange (x86)
- Use these to implement higher-level primitives
  - E.g. test-and-set on x86 (given to you for part 4) is written using compare-exchange:
    - compare_exchange(lock_t *x, int y, int z):

```
if(*x == y)
        *x = z;
        return y;
else
        return *x;
```

    - test_and_set(lock_t *l) {

```
    ?
}
```

Looking ahead: preemption

- You can start inserting synchronization code
  - disable/enable interrupts
  - atomic_test_and_set
- Where would you use these?

Synchronization

High-level
- Monitors
- Java synchronized method

OS-level support
- Special variables – mutex, futex, semaphor, condition var
- Message passing primitives

Low-level support
- Disable/enable interrupts
- Atomic instructions

- Used to implement higher-level sync primitives (in the kernel typically)
- Why not use in apps?

Semaphore review

- Semaphore = a special *variable*
  - Manipulated atomically via two operations:
    - P (wait)
    - V (signal)
- Has a counter = number of available resources
  - P decrements it
  - V increments it
- Has a queue of waiting threads
  - If execute wait() and semaphore is free, continue
  - If not, block on that waiting queue
- signal() unblocks a thread if it's waiting
- Mutex is bi-value semaphore (capacity 1)

Condition Variable

- A "place" to let threads wait for a certain event to occur while holding a lock
- It has:
  - Wait queue
  - Three functions: *wait*, *signal*, and *broadcast*
    - *wait* – sleep until the event happens
    - *signal* – event/condition has occurred. If wait queue nonempty, wake up *one* thread, otherwise *do nothing*
      - Do not run the woken up thread right away
      - FIFO determines who wakes up
    - *broadcast* – just like *signal*, except wake up all threads
  - In part 2, you implement all of these
- Typically associated with some logical condition in program

Condition Variable (2)

- **`cond_wait(sthread_cond_t cond, sthread_mutex_t lock)`**
  - Should do the following atomically:
    - Release the lock (to allow someone else to get in)
    - Add current thread to the waiters for `cond`
    - Block thread until awoken
  - Read man page for **`pthread_cond_[wait|signal|broadcast]`**
  - Must be called while holding **`lock`**! -- Why?

Semaphores vs. CVs

This slide intentionally left blank
to give you time to ponder this question deeply

Semaphores vs. CVs

Semaphores

- Used in apps
- wait() does not always block the caller
- signal() either releases a blocked thread, if any, or increases sem. counter.

Condition variables

- Typically used in monitors
- Wait() always blocks caller
- Signal() either releases blocked thread(s), if any, or the signal is lost forever.

Sample synchronization problem

Late-Night Pizza

- A group of students study for cse451 exam
- Can only study while eating pizza
- Each student thread executes the following:
  - ```
    while (must_study) {
      pick up a piece of pizza;
      study while eating the pizza;
    }
    ```
- If a student finds pizza is gone, the student goes to sleep until another pizza arrives
- First student to discover pizza is gone orders a new one.
- Each pizza has S slices.
    ```

Late-Night Pizza

- Synchronize student threads and pizza delivery thread
- Avoid deadlock
- When out of pizza, order it exactly once
- No piece of pizza may be consumed by more than one student

Semaphore / mutex solution

```
shared data:
    semaphore_t pizza;  (counting sema, init to 0, represent
                 number of available pizza resources)
    semaphore_t deliver; (init to 1)
    int num_slices = 0;
    mutex_t mutex; (init to 1) // guard updating of num_slices
```

```
Student {
   while (must_study) {
       P(pizza);
       acquire(mutex);
       num_slices--;
       if (num_slices==0)
         // took last slice
         V(deliver);
       release(mutex);
       study();
   }
}
```

```
DeliveryGuy {
    while (employed) {
        P(deliver);
        make_pizza();
        acquire(mutex);
        num_slices=S;
        release(mutex);
        for (i=0; i < S; i++)
          V(pizza);

    }
}
```

Condition Variable Solution

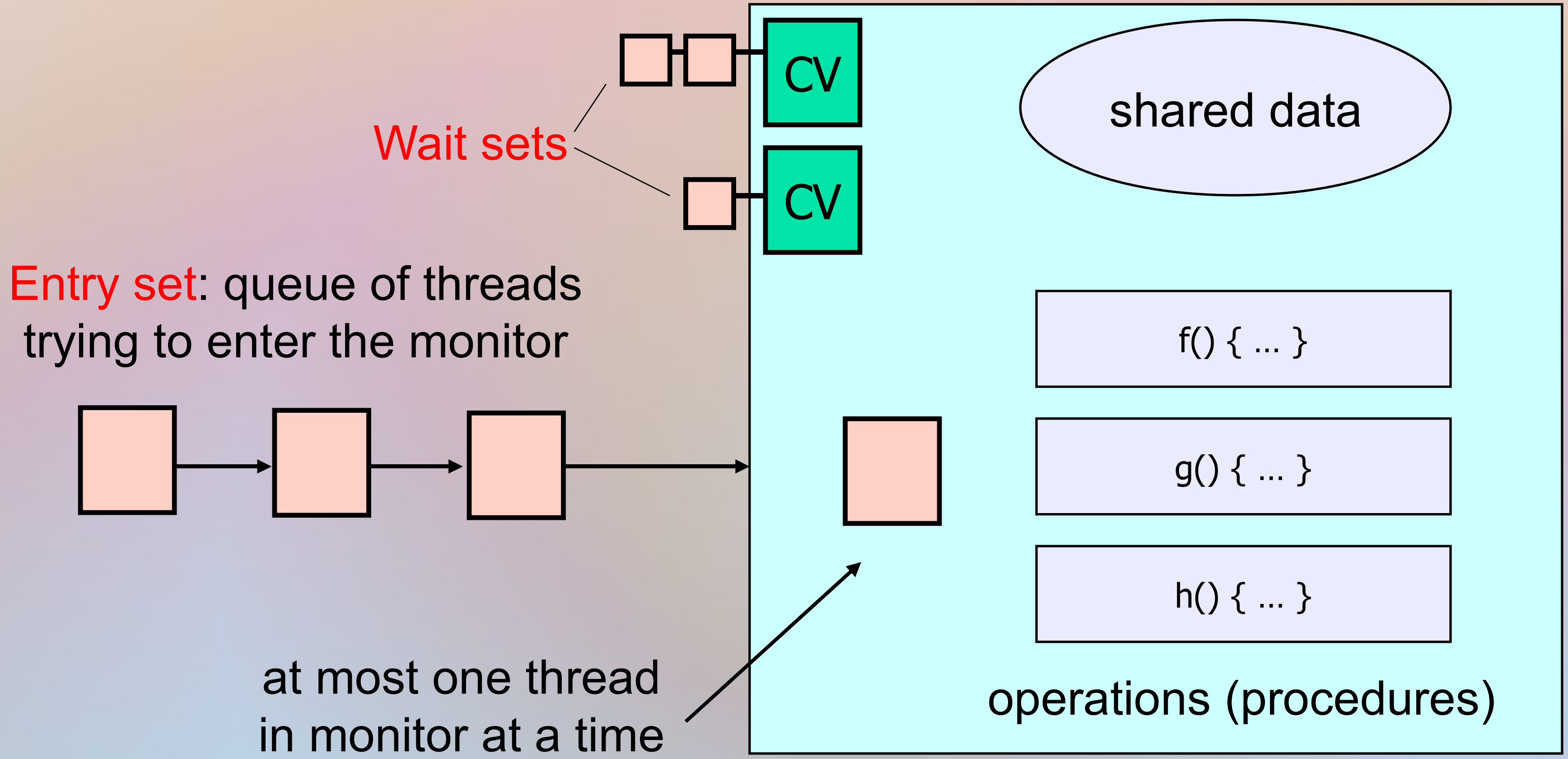
```
int slices=0;
Condition order, deliver;
Lock mutex;
bool has_been_ordered = false;
```

```
Student() {
  while(diligent) {
    mutex.lock();
    if( slices > 0 ) {
      slices--;
    }
    else {
      if( !has_been_ordered ) {
        order.signal(mutex);
        has_been_ordered = true;
      }
      deliver.wait(mutex);
    }
    mutex.unlock();
    Study();
  }
}
```

```
DeliveryGuy() {
  while(employed) {
    mutex.lock();
    order.wait(mutex);
    makePizza();
    slices = S;
    has_been_ordered = false;
    mutex.unlock();
    deliver.broadcast();
  }
}
```

Monitors: preview

- One thread inside at a time
- Lock + a bunch of condition variables (CVs)
- CVs used to allow other threads to access the monitor while one thread waits for an event to occur



Monitors in Java

- Each object has its own monitor

```
Object o
```

- The Java monitor supports two types of synchronization:
  - Mutual exclusion

```
synchronized(o) { … }
```

  - Cooperation

```
synchronized(o) { O.wait(); }
synchronized(o) { O.notify[All](); }
```