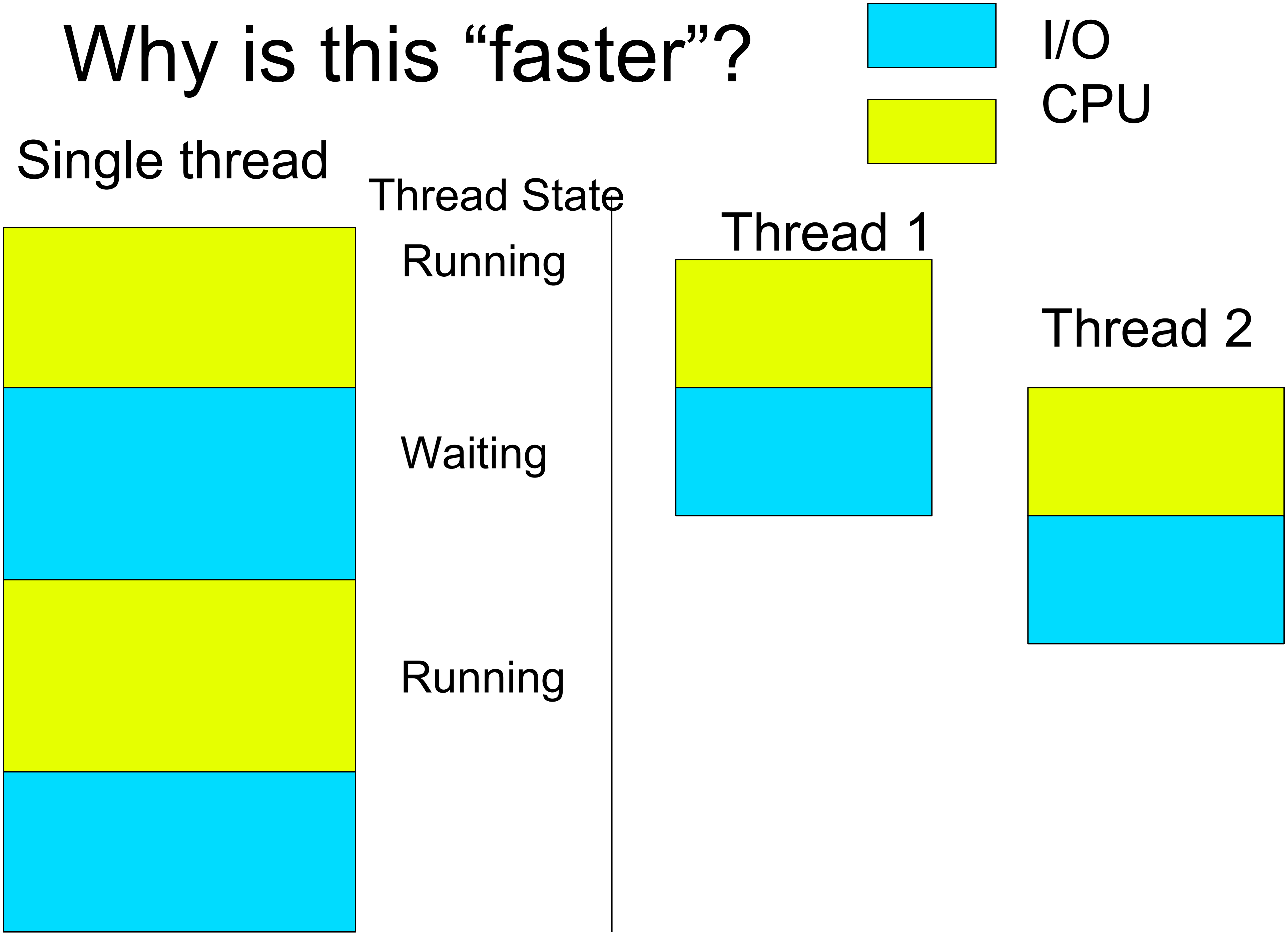
Section 4
Processes, kernel threads, user threads

Why use threads?

- Perform multiple tasks at once (reading and writing, computing and receiving input)
- Take advantage of multiple CPUs
- More efficiently use resources

Look at this silly little bouncing ball example.

Why is this “faster”?



🟦 I/O

🟨 CPU

Single thread

Thread State

Running

Waiting

Running

Thread 1

Thread 2

Quick view

- Process
  - Isolated with its own virtual address space
  - Contains process data like file handles
  - Lots of overhead
  - Every process has AT LEAST one kernel thread
- Kernel threads
  - Shared virtual address space
  - Contains running state data
  - Less overhead
  - From the OS's point of view, this is what is scheduled to run on a CPU
- User threads
  - Shared virtual address space, contains running state data
  - Kernel unaware
  - Even less overhead

Trade-offs

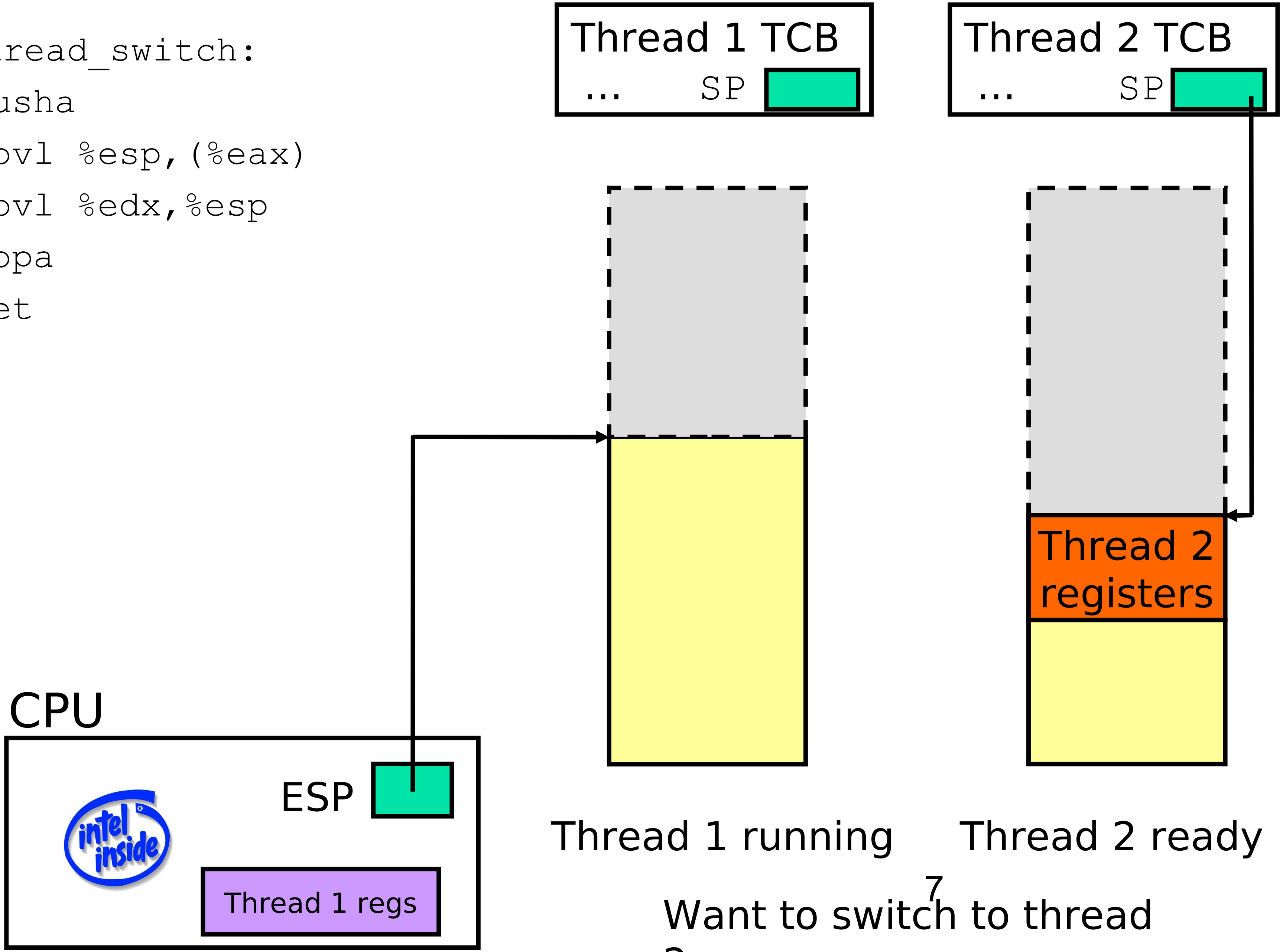
- Processes
  - Secure and isolated
  - Kernel aware
  - Creating a new process (address space!) brings lots of overhead
- Kernel threads
  - No need to create a new address space
  - No need to change address space in context switch
  - Kernel aware
  - Still need to enter kernel to context switch
- User threads
  - No new address space, no need to change address space
  - No need to enter kernel to switch
  - Kernel is unaware. No multiprocessing. I/O blocks all user threads.

When should I use which?

- Process
  - When isolation is necessary
    - Like in Chrome
- Kernel threads
  - Multiprocessor
  - heavy CPU per context switch
  - Blocking I/O
  - Compiling Linux
- User threads
  - Single processor or single kernel thread
  - Light CPU per context switch
  - Little or no blocking I/O

Context switching

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



Thread 1 TCB
… SP

Thread 2 TCB
… SP

Thread 2 registers

CPU

ESP

Thread 1 regs

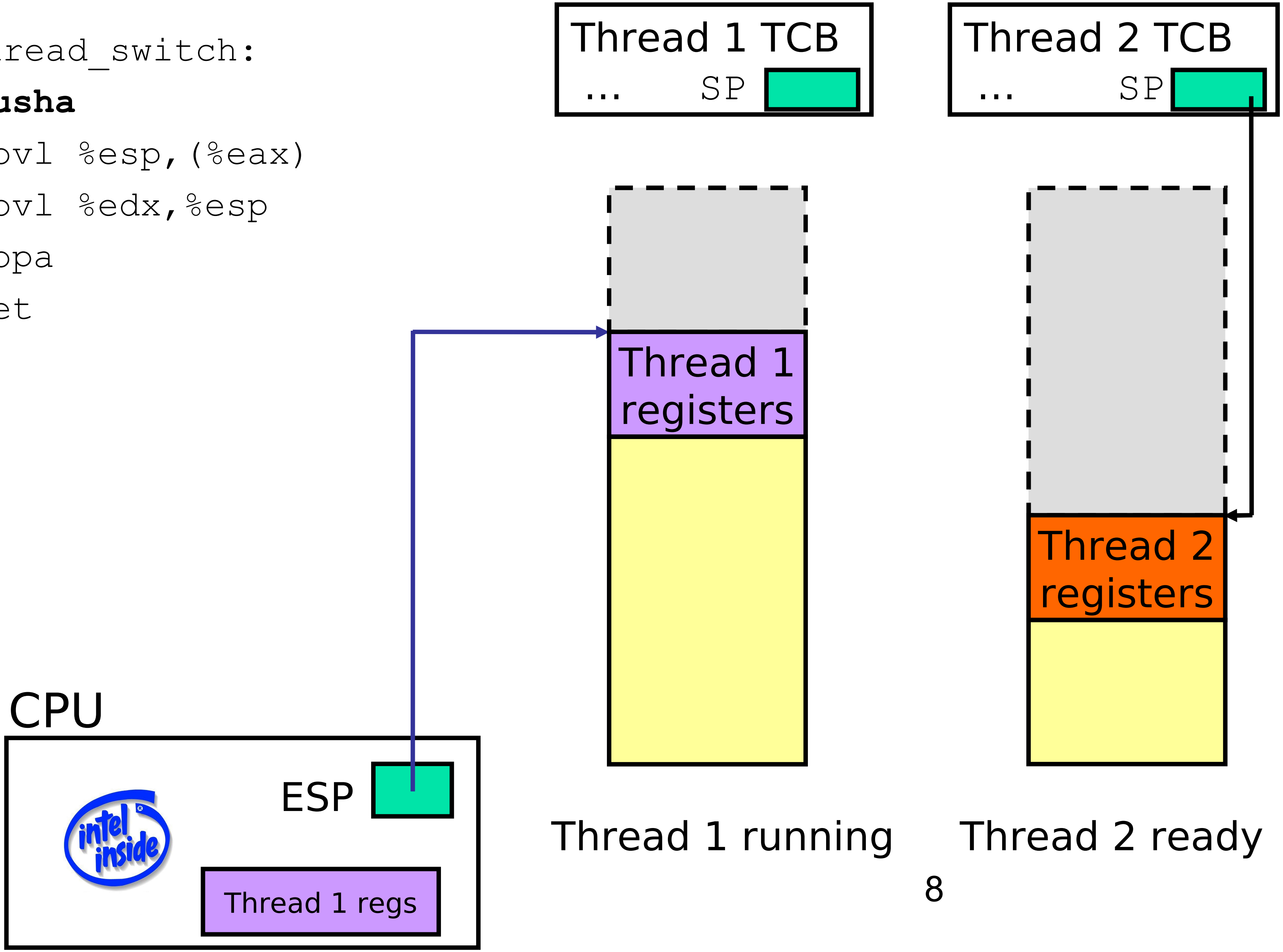
Thread 1 running

Thread 2 ready

Want to switch to thread 2…

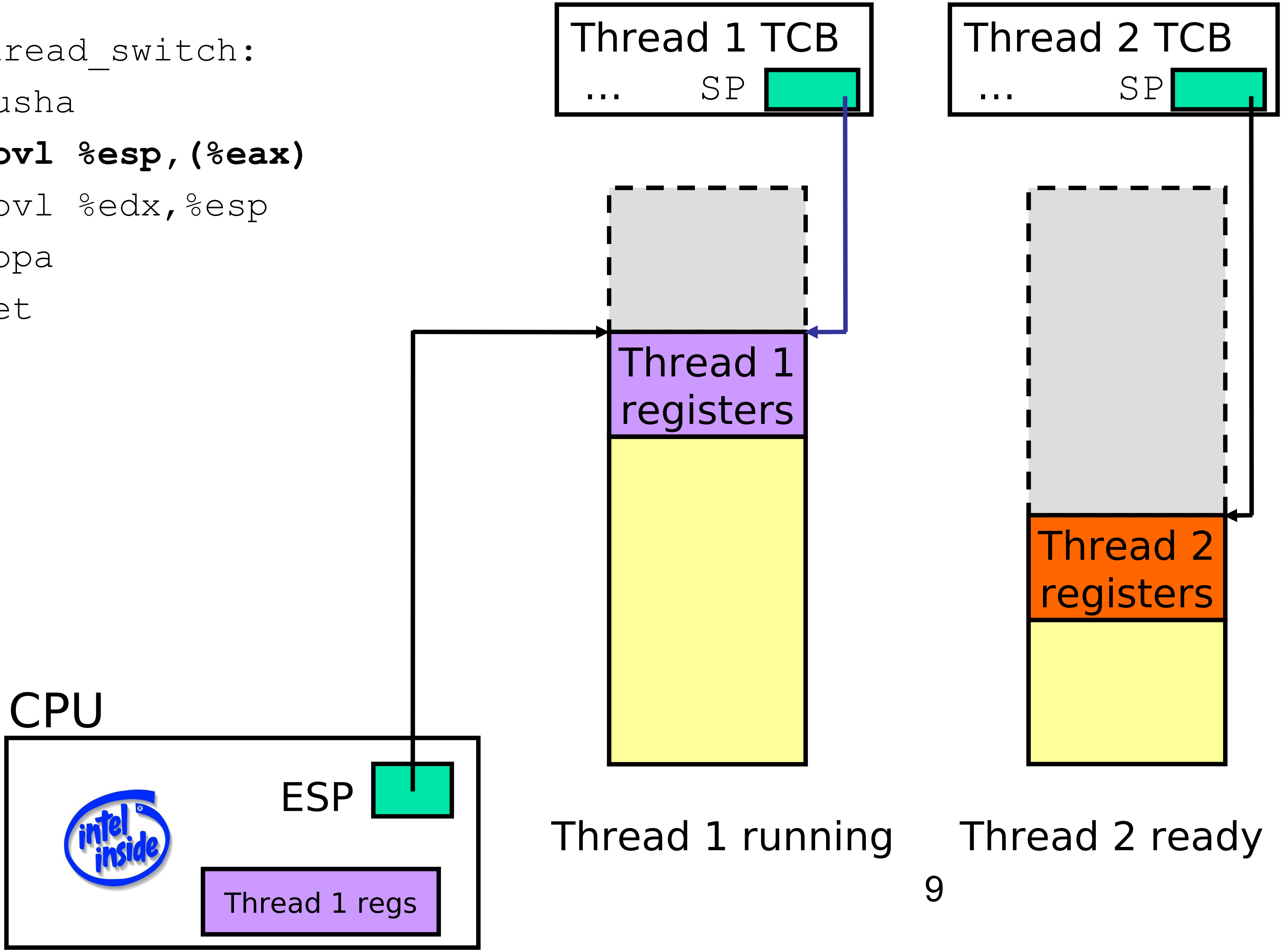
Push old context

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



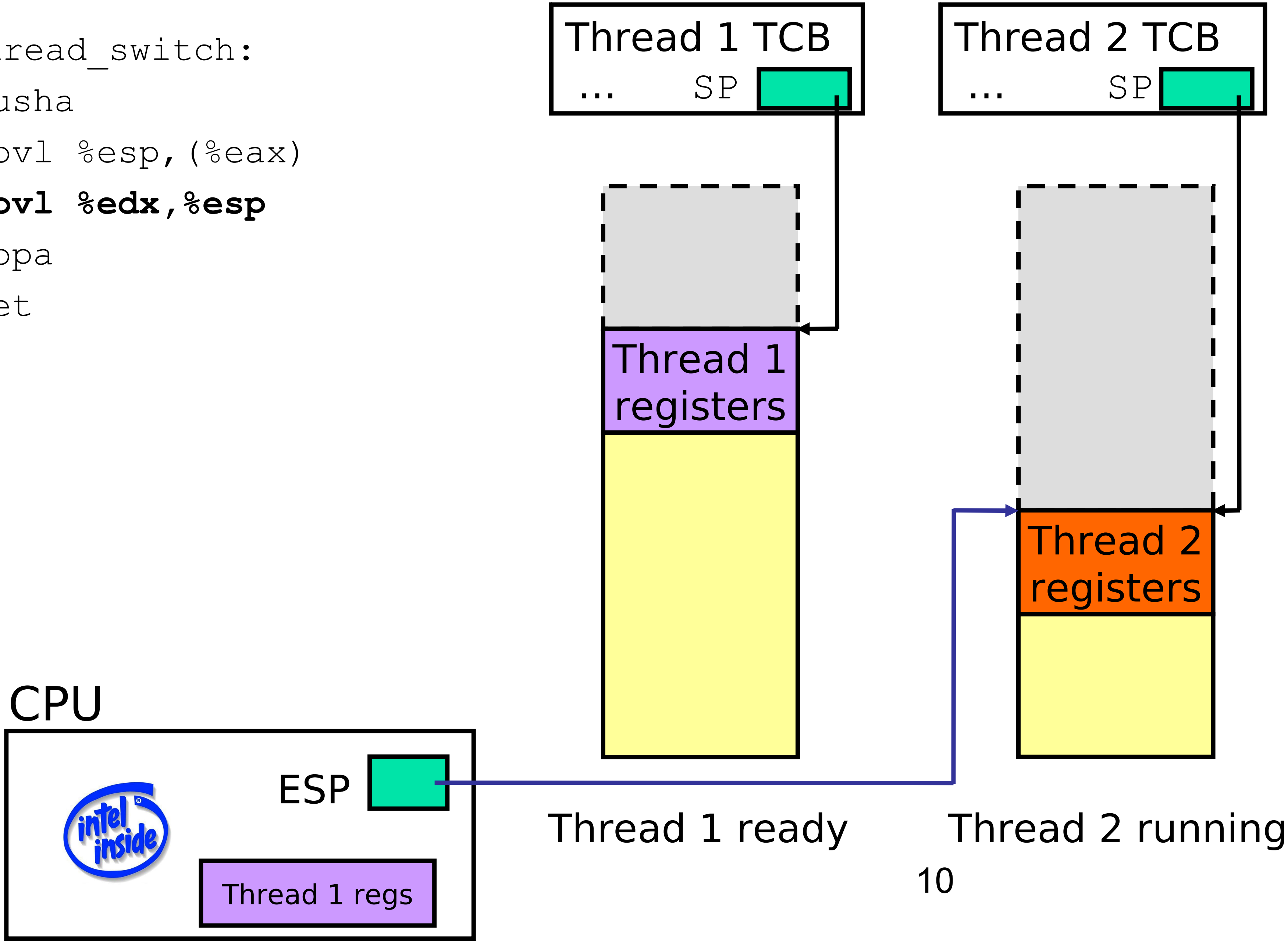
Save old stack pointer

```
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
```



Thread 1 TCB
... SP

Thread 2 TCB
... SP

Thread 1 registers

Thread 2 registers

CPU

ESP

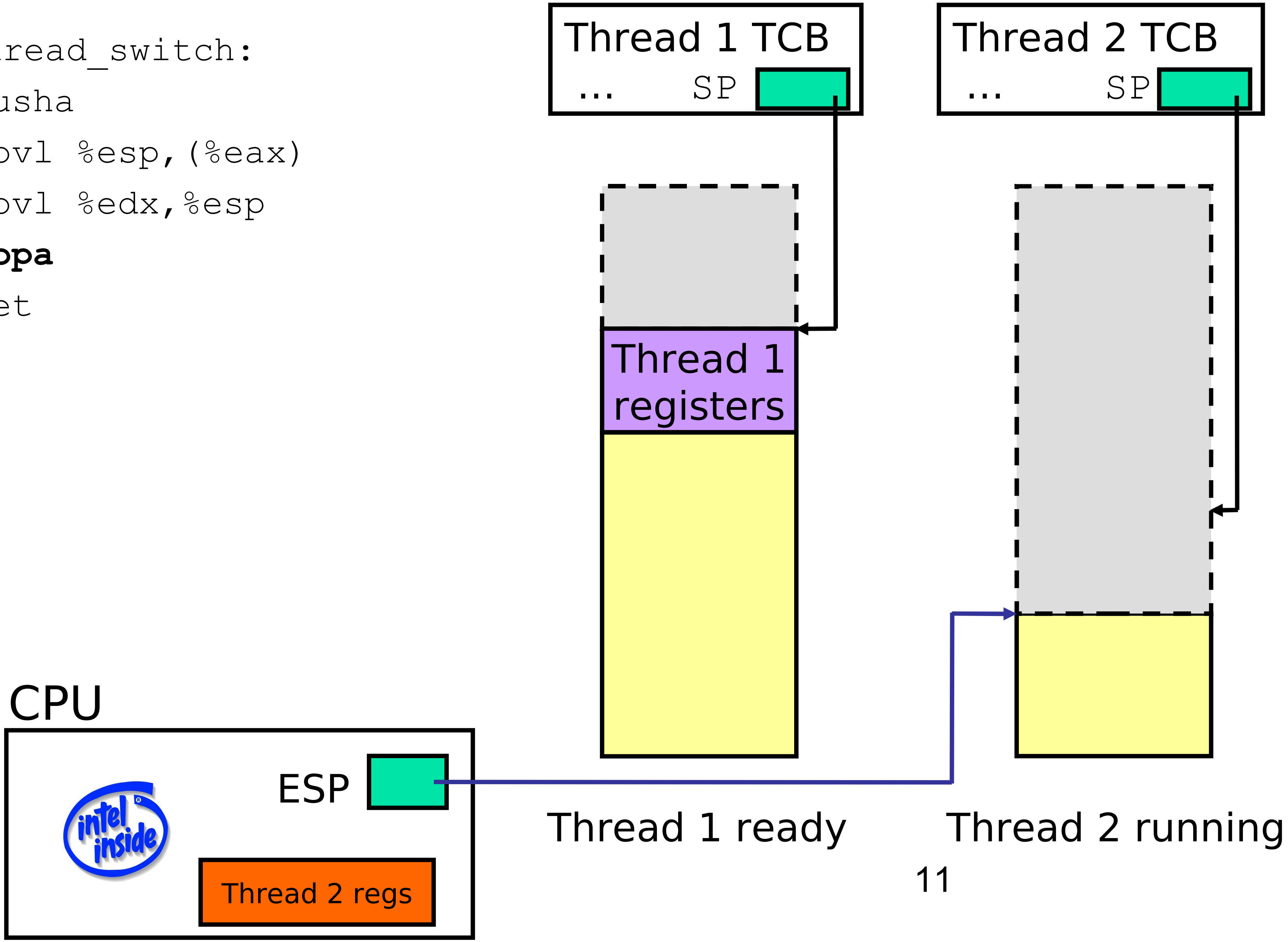
Thread 1 regs

Thread 1 ready

Thread 2 running

Pop off new context

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



Thread 1 TCB

... SP

Thread 2 TCB

... SP

Thread 1 registers

CPU

ESP

Thread 2 regs

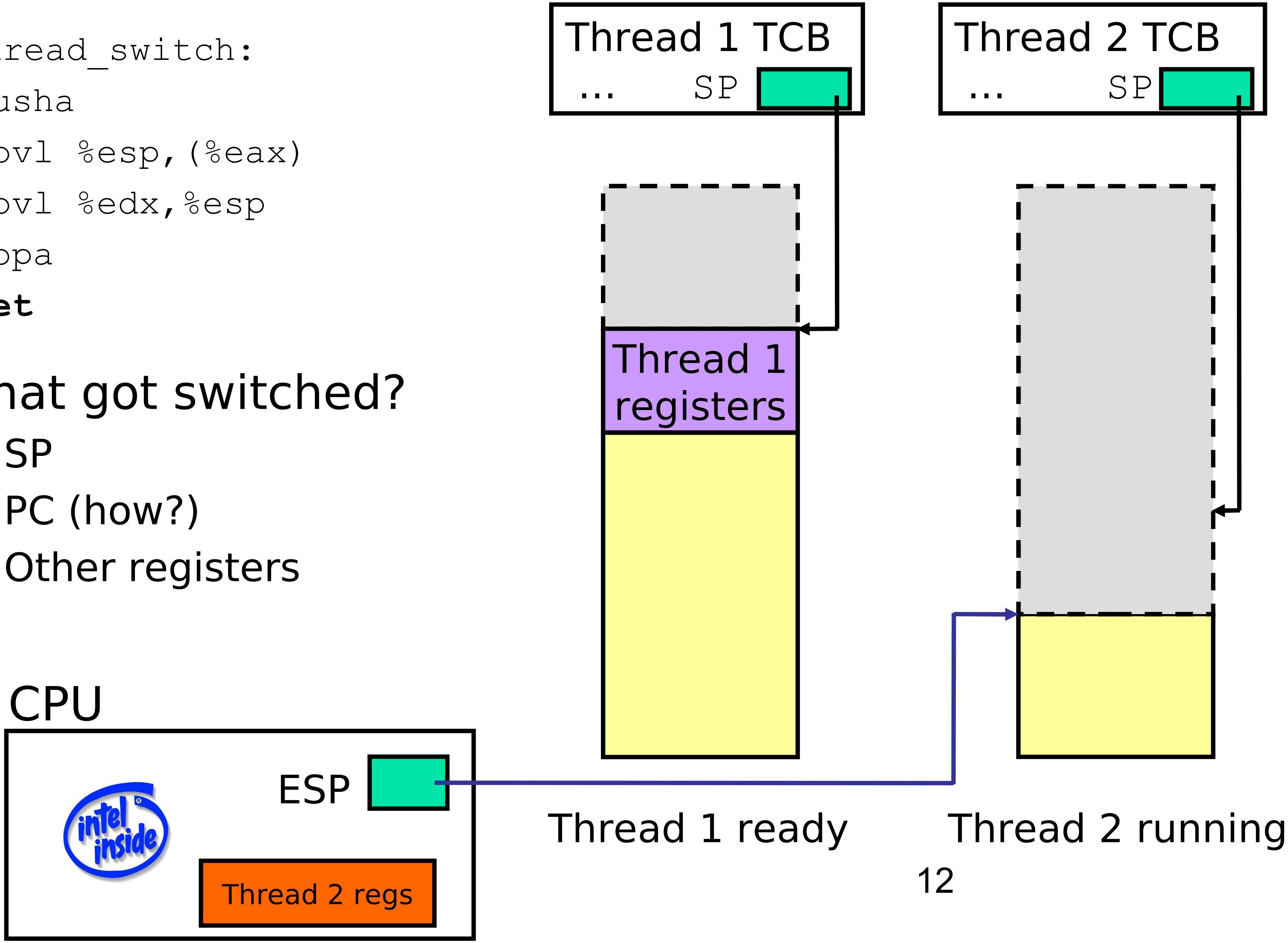
Thread 1 ready

Thread 2 running

Done; return

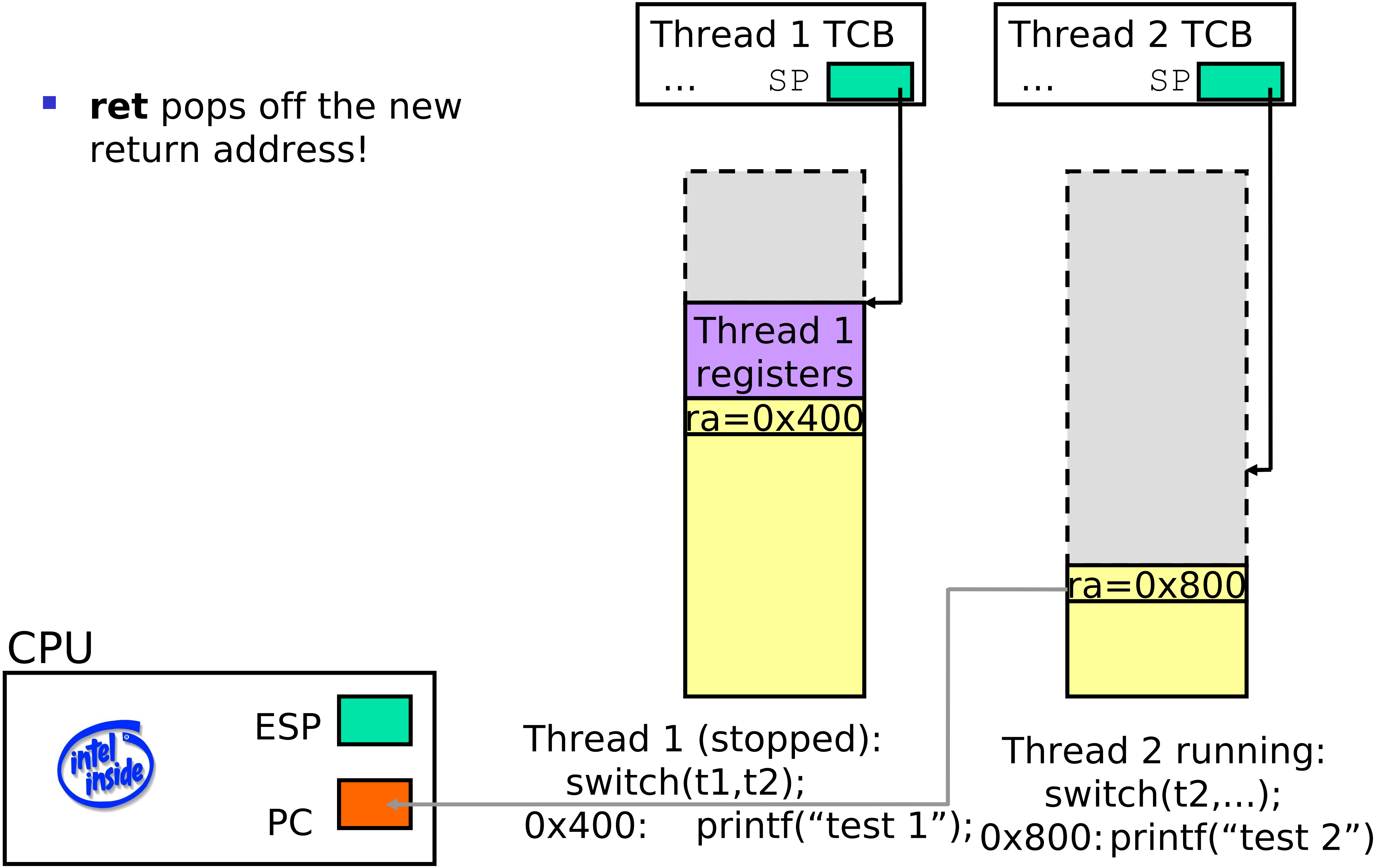
```
Xsthread_switch:
  pusha
  movl  %esp,(%eax)
  movl  %edx,%esp
  popa
  ret
```

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



Adjusting the PC

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



Thread 1 TCB

… SP

Thread 2 TCB

… SP

Thread 1 registers

ra=0x400

ra=0x800

CPU

ESP

PC

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

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

Context Switching

- So was this for kernel threads or user threads?
    - Trick question! This can be accomplished in either kernel or user mode.

Theading Models

Between kernel and user threads, a process might use one of three models:

- One to one (1:1)
  - Only use kernel threads without user level threads on top of them.
- Many to one (M:1)
  - Use only one kernel thread with many user level threads built on top of them.
- Many to Many (N:M)
  - Use many kernel threads with many user level threads.

Threading Models

- Many to many sounds nice, intuitively but...
  - ...it can actually get problematic in its complexity
  - See Scheduler Activations

- Linux actually runs One to one
- Windows runs a lazy version of Scheduler Activations.

Schedules

- Make sure you understand the metrics
  - maximize CPU utilization
  - maximize throughput (requests completed / s)
  - minimize average response time (average time from submission of request to completion of response)
  - minimize average waiting time (average time from submission of request to start of execution)
  - And starvation/fairness
  - And which schedules maximize which metrics

Linux Scheduler

- Completely Fair Scheduler (CFS)
    - Linux's scheduler since 2.6.23
    - Computes the fair CPU share for a task
        - Based on current number of tasks for a user
    - Tracks the difference between run time and ideal fair share
    - Schedules longest waiting non-real-time task
        - Implemented in red-black tree
- But, is this really fair?