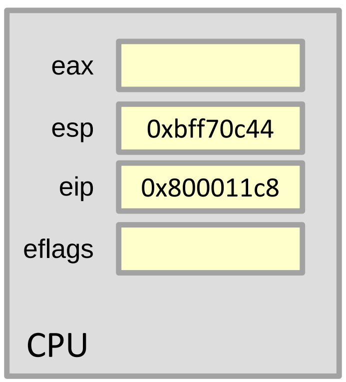
CSCI 350
Ch. 4 – Threads and Concurrency

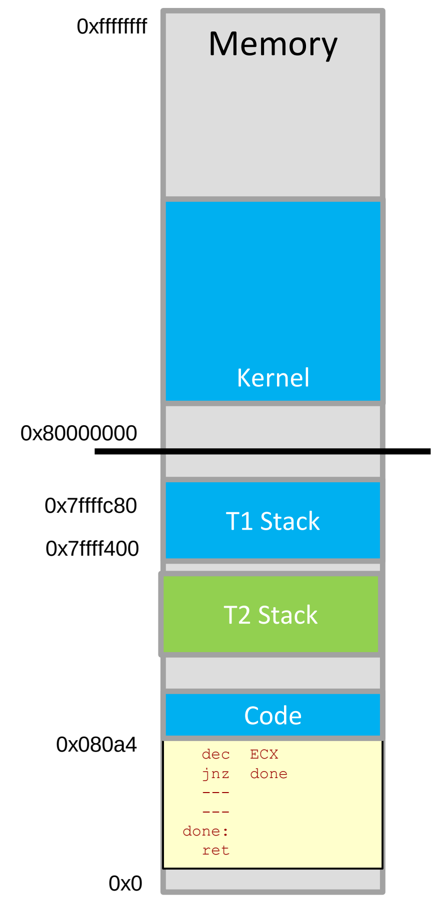
Mark Redekopp
Michael Shindler & Ramesh Govindan

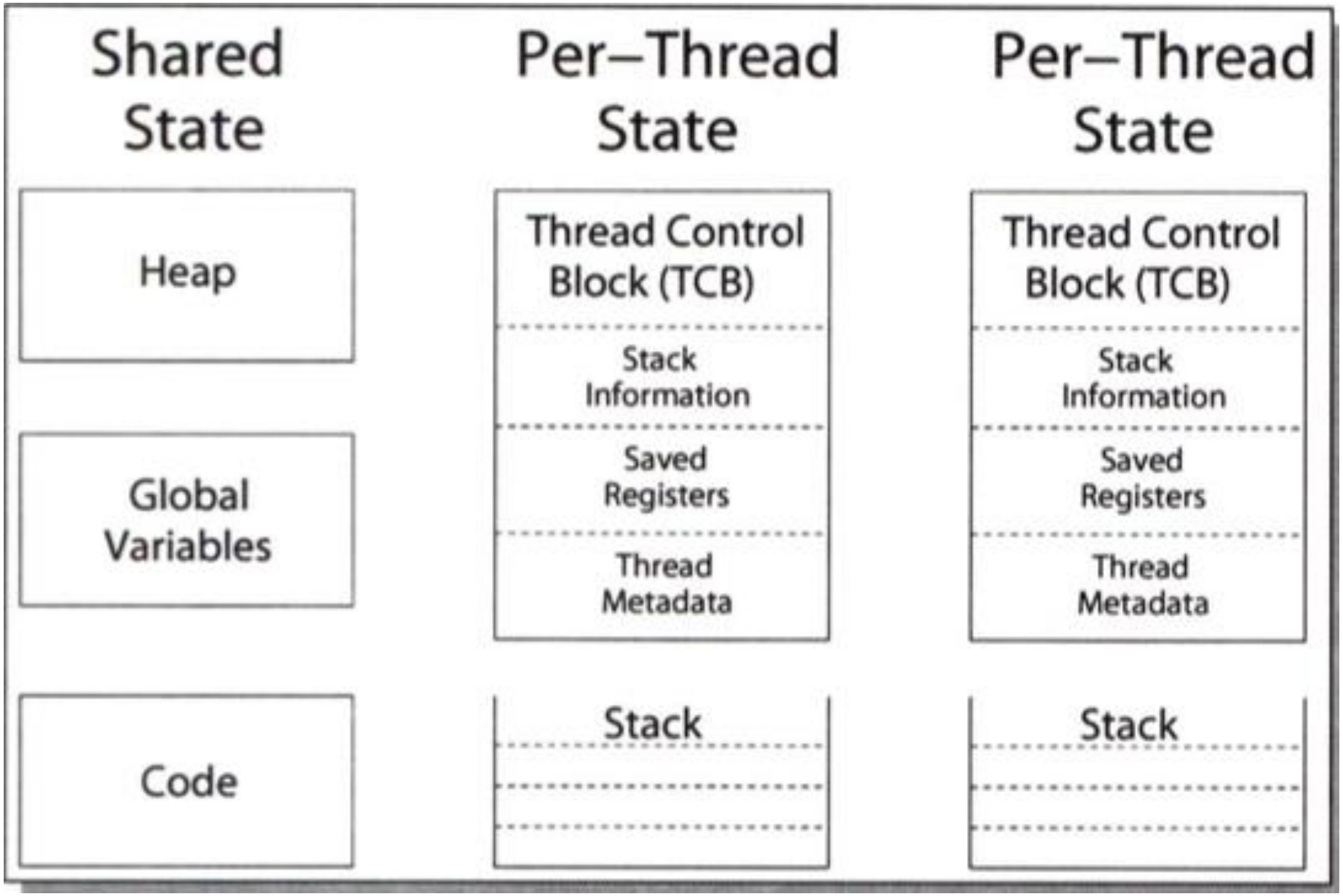
WHAT IS A THREAD AND WHY USE THEM

What is a Thread?

- **Thread (def.)*:*** Single execution sequence representing a separately schedulable task
  - Execution sequence: Registers, PC (IP), Stack
  - Schedulable task: Can be transparently paused and resumed by the OS scheduler

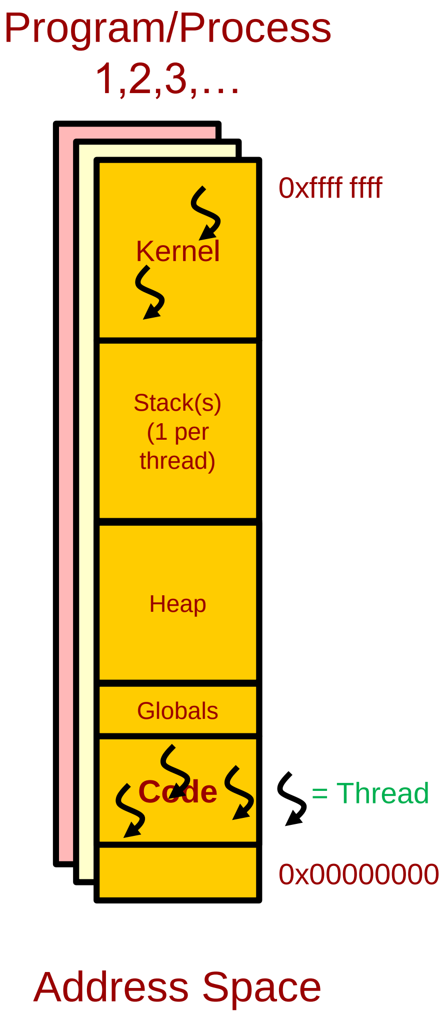





Shared State
Heap
Global Variables
Code
Per–Thread State
Thread Control Block (TCB)
Stack Information
Saved Registers
Thread Metadata
Stack
Per–Thread State
Thread Control Block (TCB)
Stack Information
Saved Registers
Thread Metadata
Stack

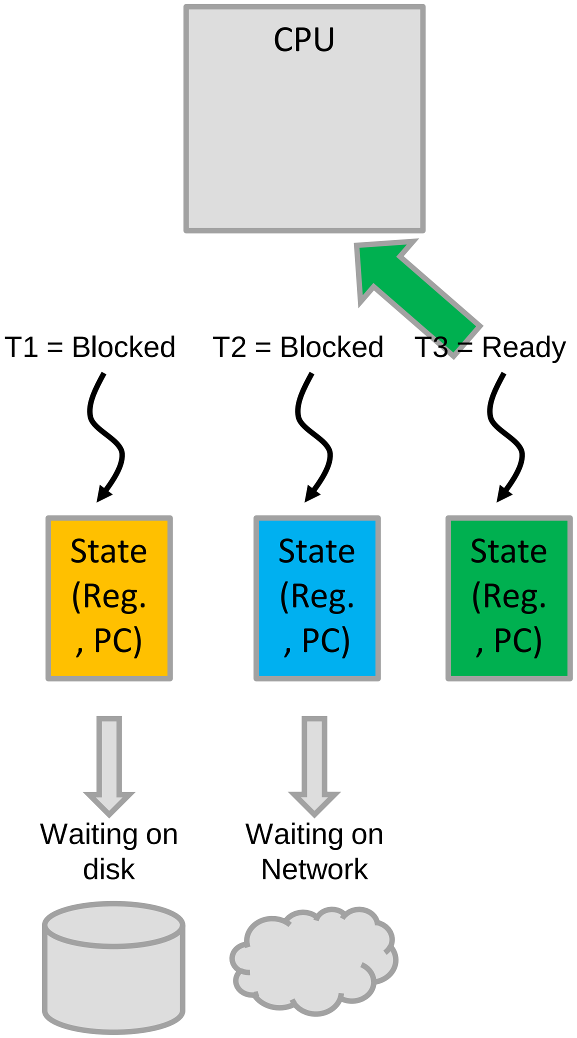
Threads vs. Processes

- **Process (def.): Address Space + Threads**
  - Address space is protected from other processes
  - 1 or more threads
    - Pintos, Original Unix: 1 Process = 1 Thread
    - Most OSs: 1 Process = n Threads
- Kernel may have many threads and can access any processes memory



Why Use Threads?

- Unit of parallelism
  - Take advantage of multiple cores
  - Increase utilization of single-core
    - In case of long-latency events (namely I/O) where one thread must wait, let the processor execute another thread
- Hard (if not impossible) to express concurrency in a single thread
  - See example of e-mail client on next slide



Email Client (Threaded vs. Non-Threaded)

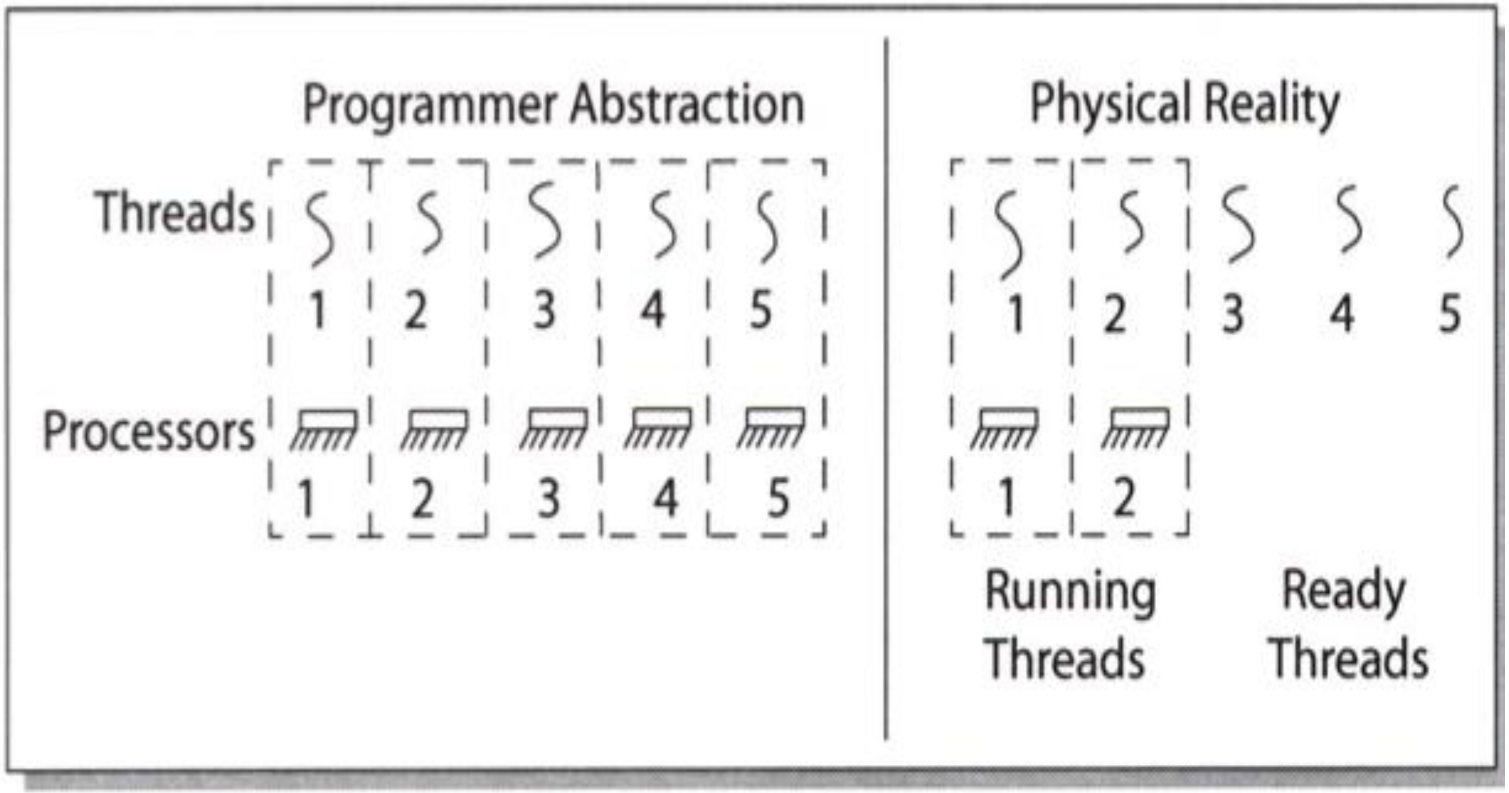
```
/* Thread 1 */
void searchEmail(List* results,
                 char* target)
{
  for(i=0; i < numEmails; i++)
    if(contains(emails[i], target))
      results->push_back(emails[i]);
}

/* Thread 2 */
void checkIncoming(bool* newMsg)
{
  while(1){
   fd_set rset;
   FD_ZERO(&rset);
   FD_SET(sockID, &rset);
   uint64_t msTimeOut = 1000; // milli
   select(FD_SETSIZE, rfds, ..., msTimeOut);
   *newMsg = FD_ISSET(sockID, &rset);
 }
}

/* Thread 3 */
void checkAndHandleUserInput()
{
  while(1){
    if(pressCompose()) { ... }
    else if(pressDeleteMsg()) { ... }
    else {... }
  }
}
```

```
void doItAll( /* args */ )
{
  int si = -1, checkCnt = 100;
  while(1){
     if(startSearch()) si = 0;
     if(si != -1) {
       /* Search next email */
       if(contains(emails[si], target))
         results->push_back(emails[si]);
       if(++si == numEmails);
     }
     /* Check new msgs every 100th itr */
     if(--checkCnt == 0){
        checkCnt = 100;
        uint64_t msTimeOut = 0; // none
        select(..., msTimeOut);
        *newMsg = FD_ISSET(...);
     }
     if(pressCompose()) { ... }
     else if(pressDeleteMsg()) { ... }
     else {... }
  }
}
```

- Left: Natural way of expressing concurrent tasks as separate entities and sequences of execution
- Right: Attempt to ensure response times among the tasks with only 1 thread

Main Idea

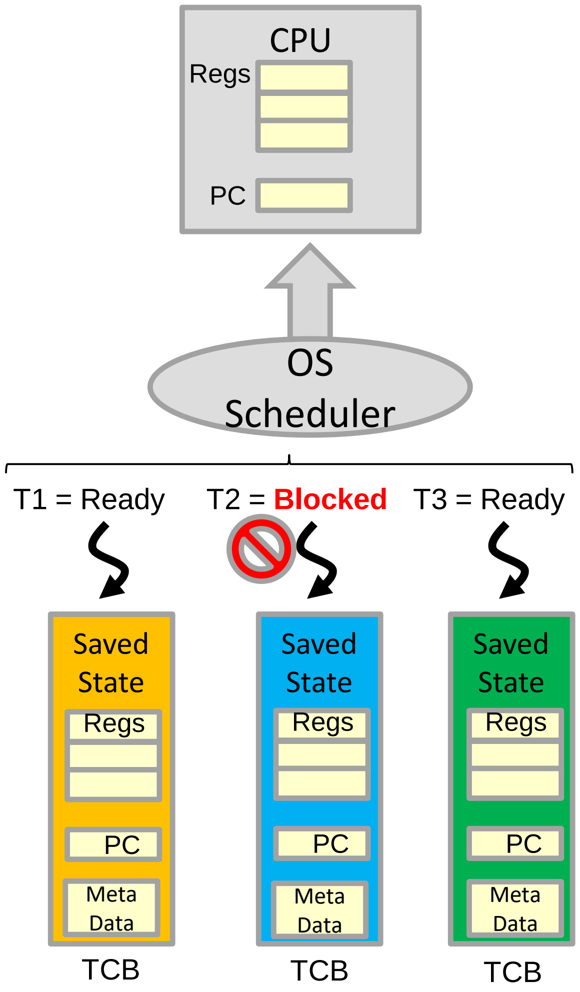
- **Key idea**: Operating system multiplexes these threads on the available processors *by suspending and resuming threads transparently*
- A thread provides a **virtualization** of the processor (i.e. nearly infinite number of "processors")
  - Number of threads >> number of processors


Programmer Abstraction
Threads
Processors
1
2
3
4
5
Physical Reality
Running
Threads
Ready
Threads

SCHEDULING AND INTERLEAVING

OS Scheduler & Context Switches

- A primary OS component is the scheduler
  - Chooses one of the "ready" threads and grants it use of the processor
  - Saves the state (registers + PC) of the previously executing thread and then restores the state of the next chosen thread
  - **Swapping threads (saving & restoring state) is known as a context switch**
  - Appears transparent to the actual thread code
- Policies for choosing next thread are examined in a subsequent chapter (for now assume simple round-robin / FIFO)
- Threads have memory to store register, PC, and some metadata (thread ID, thread-local variables, etc.) in some kind of OS data structure usually called a thread control block (TCB)



When to Context Switch

- Cooperative Multitasking (Multithreading)
  - Current running thread gets to determine (voluntarily) when it will yield the processor
  - Used in some older OSs (e.g. Windows 3.1)
- Preemptive Multitasking (Multithreading)
  - OS can unilaterally cause the current running thread to be context switched
  - Generally done based on some regular timer interval (i.e. time quantum) such as every 10ms
  - Used in most OSs

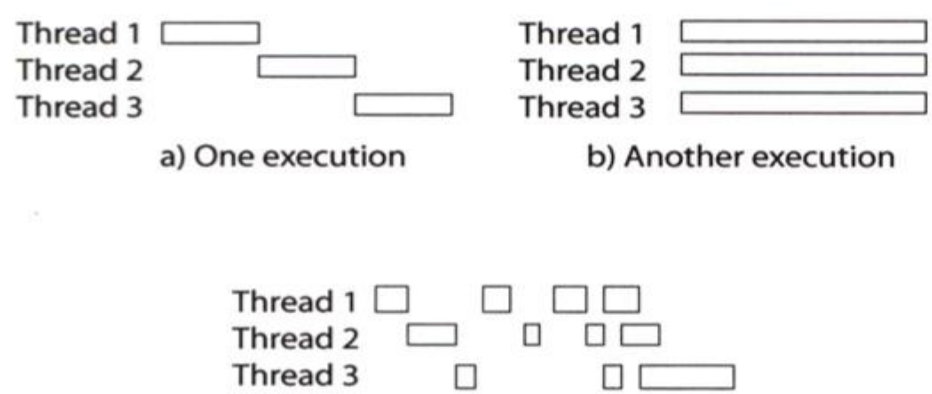
Interleavings

- Generally, threads can be interleaved (i.e. swapped) at arbitrary times by the OS
  - Exception 1: certain situations in a real-time OS
  - Exception 2: Kernel explicitly disables interrupts temporarily
- The programmer MUST NOT assume any particular interleaving or speed of execution
  - Ensure correctness in the worst possible case (i.e. context switch at the most vulnerable time)
  - Assume "variable" rate of execution
    - No idea when cache miss or page fault will occur
    - Even in absence of these, speed of execution of code is not constant (due to pipelining, branch prediction, etc.)

Race Condition

- A *race condition* occurs when the behavior of the program depends on the interleaving of operations of different threads.
- Example: Assume x = 2
  - T1: x = x + 5
  - T2: x = x * 5
- Outcomes
  - Case 1: T1 then T2
    - After T1: x = 7
    - After T2: x = 35
  - Case 2: T2 then T1
    - After T2: x = 10
    - After T1: x = 15
  - Case 3: Both read before either writes, T2 Write, T1 Write
    - x = 7
  - Case 4: Both read before either writes, T1 Write, T2 Write
    - x = 10

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		y = y + x	....
		z = x + 5y	z = x + 5y



a) One execution

b) Another execution

c) Another execution

Critical Section: First Look

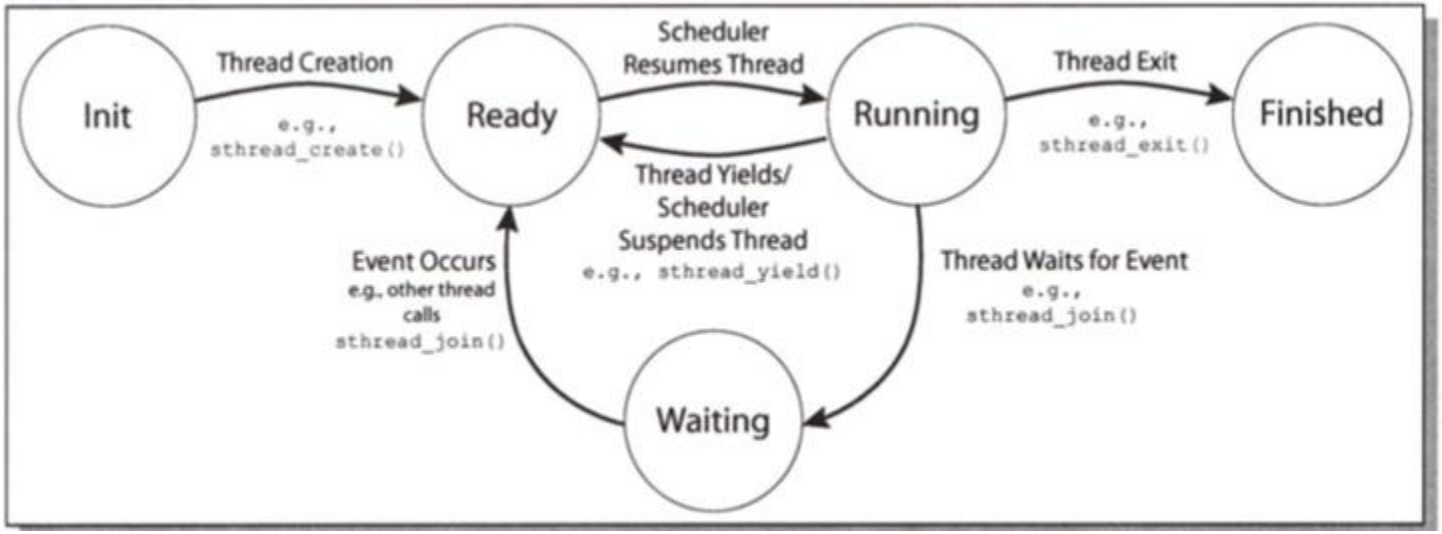
- A **critical section** is a section of code that should be performed without the chance of context switching in the middle (i.e. updating certain OS data structures)
- On a single-processor system one way to ensure no context switch is to disable interrupts
  - Now timer or other interrupt cannot cause the current thread to be context switched
- General pattern:

```
old_state = getInterruptStatus();
disableInterrupts();
/* Do critical task */
setInterrupts(old_state);
```

- Why do we need old_state and not just enableInterrupts() at the end?

Thread Scheduling State

- Two kinds of thread state:
  - It's current register, PC, stack values
  - It's scheduling status
    - I'll refer to this as its scheduling state
- Scheduling states
  - INIT: Being created
  - READY: Able to execute and use the processor
  - RUNNING: Currently running on a processor
  - BLOCKED/WAITING: Unable to use the processor (waiting for I/O, sleep timer, thread join, or blocked on a lock/semaphore)
  - FINISHED: Completed and waiting to be deleted/deallocated
    - We can't delete the TCB and especially the stack in the context of the dying thread (we need the stack to know where to return)
    - Instead, we list it as a finished thread and the scheduler can come and clean it up as it schedules the next thread



State of Thread	Location of Thread Control Block (TCB)	Location of Registers
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB

THREADING API

Common Thread API

- thread_create
- thread_yield
- thread_join
- thread_exit
- thread_sleep

```
#include <stdio.h>
#include "sthread.h"

static void go(int n);

#define NTHREADS 10
static sthread_t threads[NTHREADS];

int main(int argc, char **argv)
{
  int ii;

  for(ii = 0; ii < NTHREADS; ii++){
    sthread_create(&(threads[ii]), &go, ii);
  }
  for(ii = 0; ii < NTHREADS; ii++){
    long ret = sthread_join(threads[ii]);
    printf("Thread %d returned %ld\n", ii, ret);
  }
  printf("Main thread done.\n");
  return 0;
}

void go(int n)
{
  printf("Hello from thread %d\n", n);
  sthread_exit(100 + n);
  // Not reached
}
```

Note: On a multicore many thread libraries allow a thread to specify an **processor affinity** indicating which processor it prefers to run on.

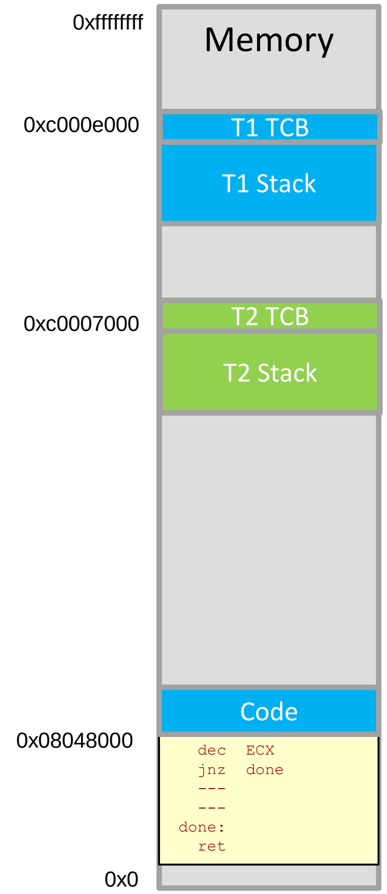
Review Questions

- Why use threads? What benefits do they provide over traditional serial execution?
- As the programmer, how do you know if your program has a race condition? Where would you start to debug a race condition?

OS BOOKKEEPING & THREAD METADATA

Thread Control Block

- Per-thread state maintained by the OS
  - Scheduling state, priority
  - Last Stack Pointer
  - Registers/PC can be stored in TCB or on stack (Pintos places them on the stack)
- TCBs can be stored in some kernel list
  - Pintos places TCB at the base of the stack



Pintos TCB

```
enum thread_status
  {
    THREAD_RUNNING,     /* Running thread. */
    THREAD_READY,       /* Not running but ready to run. */
    THREAD_BLOCKED,     /* Waiting for an event to trigger. */
    THREAD_DYING        /* About to be destroyed. */
  };

struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

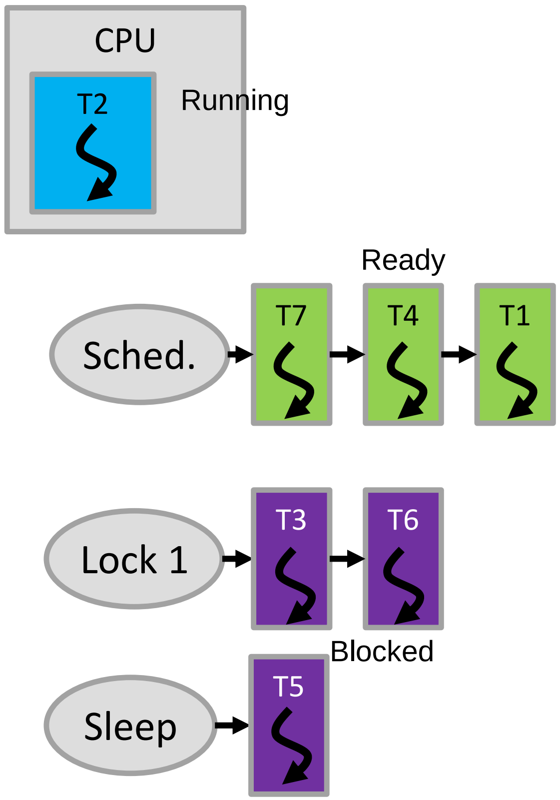
Linux

```
struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
  atomic_t usage;
  unsigned int flags; /* per process flags, defined below */
  unsigned int ptrace;

#ifdef CONFIG_SMP
  struct llist_node wake_entry;
  int on_cpu;
  struct task_struct *last_wakee;
  unsigned long wakee_flips;
  unsigned long wakee_flip_decay_ts;
  int wake_cpu;
#endif
  int on_rq;
  int prio, static_prio, normal_prio;
  unsigned int rt_priority;
  /* And a lot more!!! */
}
```

Process/Thread Control Block (task_struct):
/usr/src/linux-headers-3.13.0-24-generic/include/linux/sched.h:1042

Where's The Thread

- The OS must keep track of threads
- Can maintain a list of all threads but each thread may be in a different state
- Generally, thread can be in a "ready list" that the scheduler will choose from on a context switch
  – Running thread can either be the head of this list (Linux) or not in this list at all (Pintos)
- Other threads can be blocked. Blocked on what?
  – Sleep timer
  – Lock, Cond. Var., Semaphore



Resources & Examples

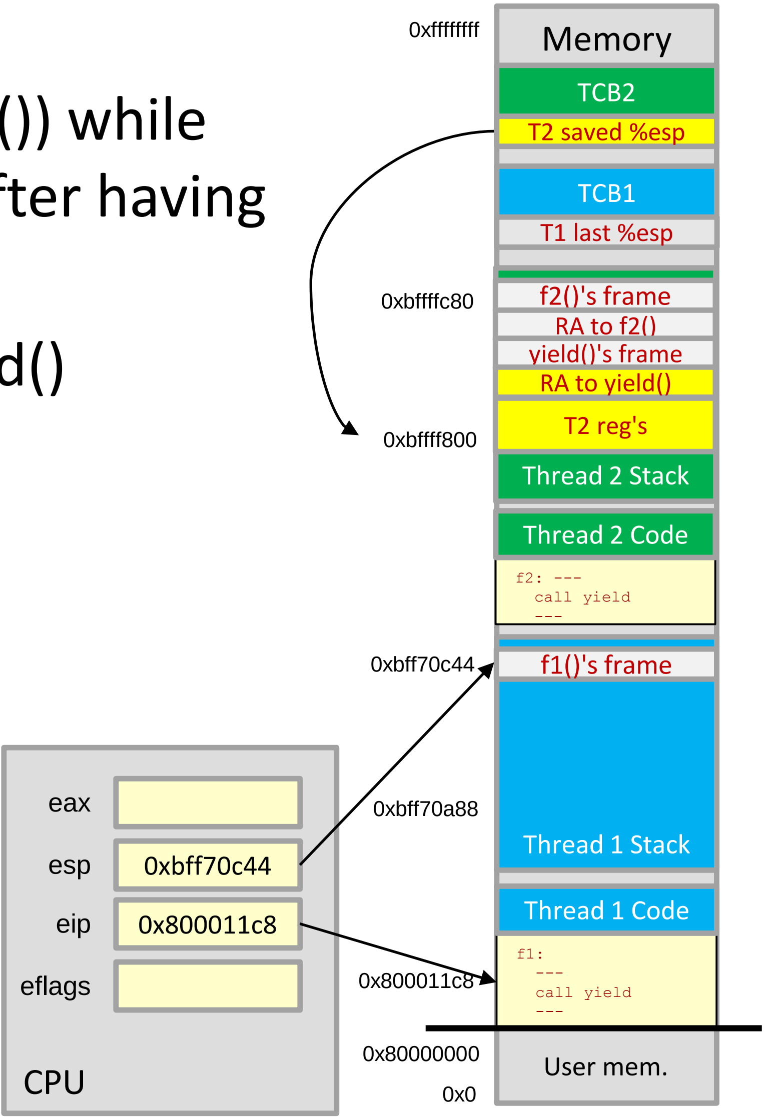
- Process Control Block (task_struct): /usr/src/linux-headers-3.13.0-24-generic/include/linux/sched.h
  - Around line 1042
- Syscalls: http://man7.org/linux/man-pages/man2/syscalls.2.html
  - Defined in <unistd.h>

In-Depth

THREAD CONTEXT SWITCH

Thread Context Switch Example

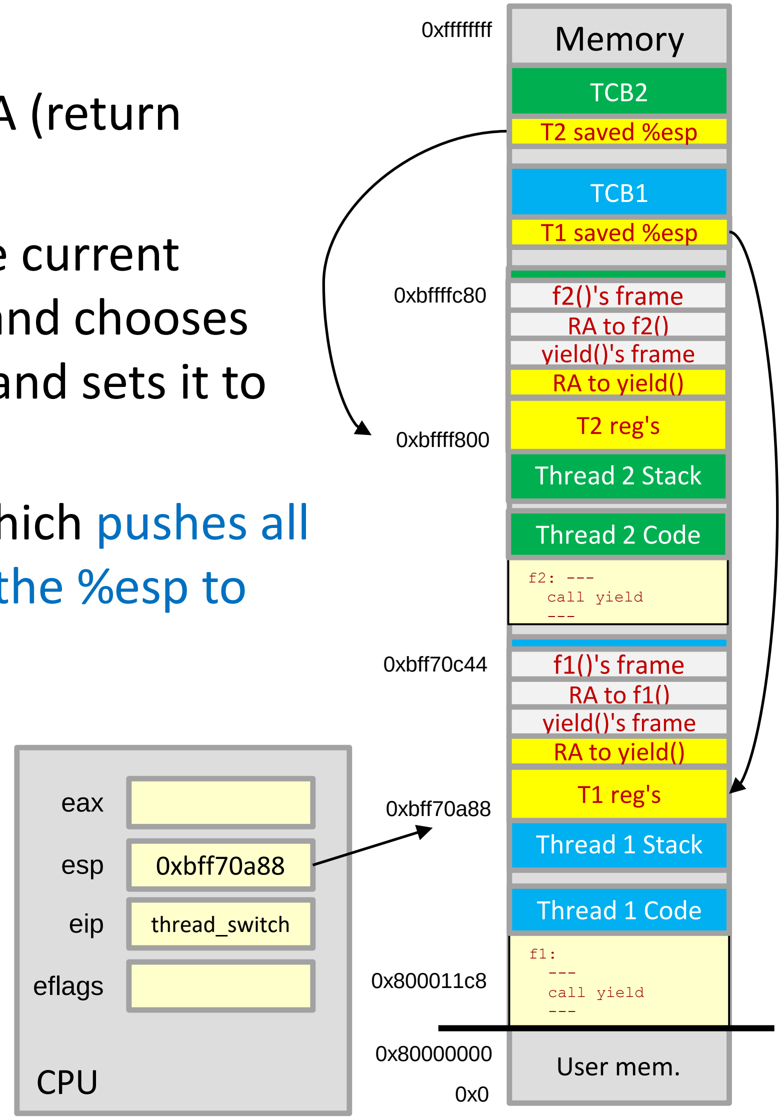
- Thread 1 is currently executing (in f1()) while thread 2 is waiting in the ready list after having yielded the CPU in f2()
- Thread 1 is about to call thread_yield()



Thread Context Switch Example

- Thread 1 calls thread_yield() pushing the RA (return address) to f1() on the stack
- thread_yield() disables interrupts, adds the current thread to the waiting list (in state READY), and chooses the next thread to schedule (i.e. Thread 2) and sets it to RUNNING
- thread_yield() then calls thread_switch() which pushes all T1's registers onto its stack and then saves the %esp to TCB1

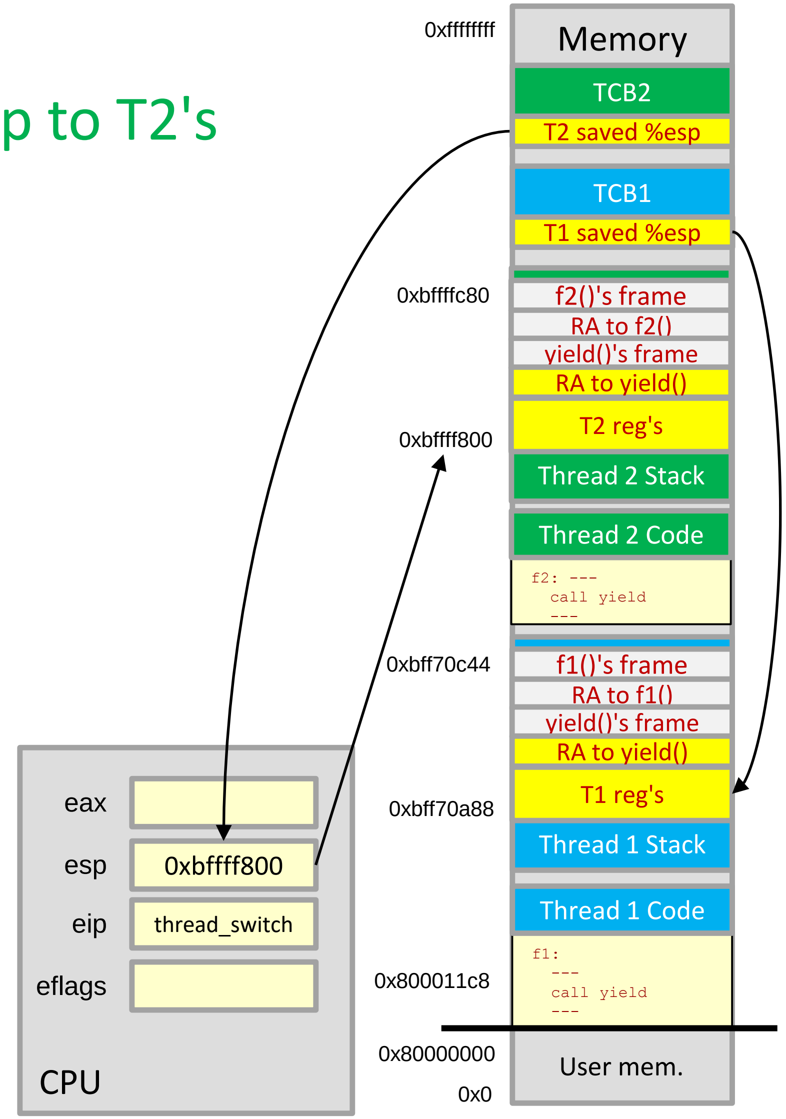
```
thread_switch:
   # Note that the SVR4 ABI allows us to
   # destroy %eax, %ecx, %edx,
   pushl %ebx
   pushl %ebp
   pushl %esi
   pushl %edi

   # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
   mov thread_stack_ofs, %edx
   # Save current stack pointer to old thread's stack
   movl SWITCH_CUR(%esp), %eax
   movl %esp, (%eax,%edx,1)...
```



Thread Context Switch Example

- thread_switch() then resets the %esp to T2's saved version from TCB2

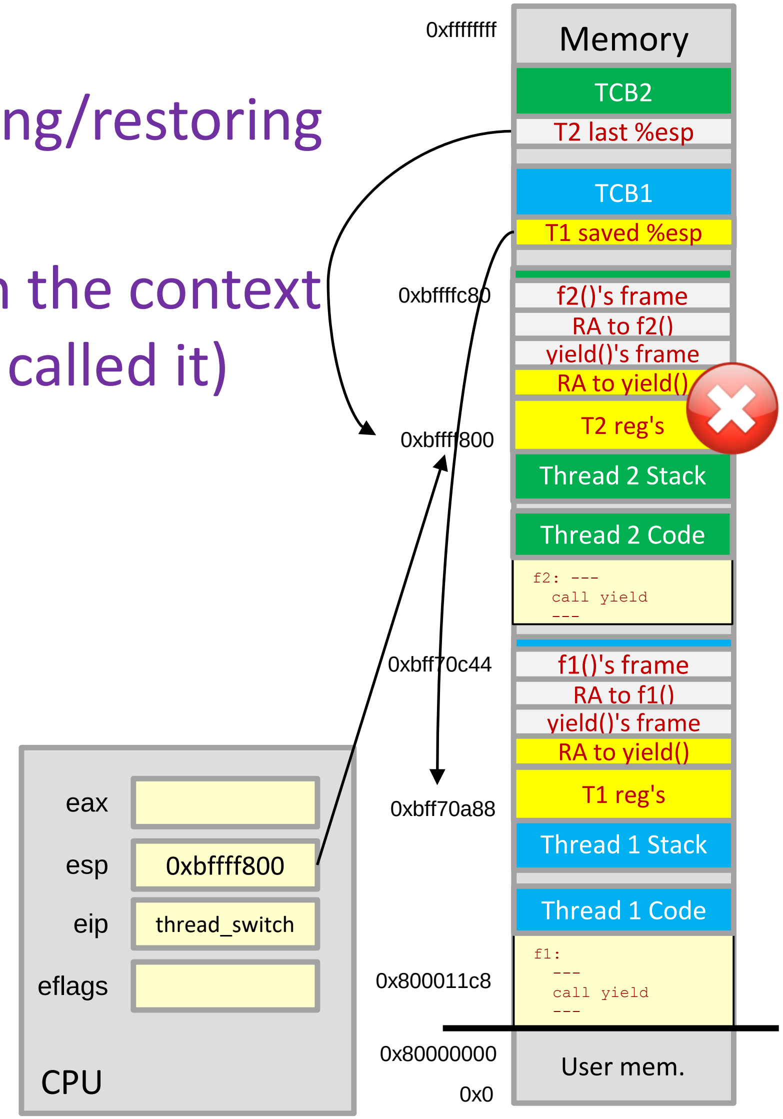
```
thread_switch:
   # Note that the SVR4 ABI allows us to
   # destroy %eax, %ecx, %edx,
   pushl %ebx
   pushl %ebp
   pushl %esi
   pushl %edi

   # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
   mov thread_stack_ofs, %edx

   # Save current stack pointer to old thread's stack
   movl SWITCH_CUR(%esp), %eax
   movl %esp, (%eax,%edx,1)
   # Restore stack pointer from new thread's stack.
   movl SWITCH_NEXT(%esp), %ecx
   movl (%ecx,%edx,1), %esp

   # Restore caller's register state.
   popl %edi
   popl %esi
   popl %ebp
   popl %ebx
   ret
```



Thread Context Switch Example

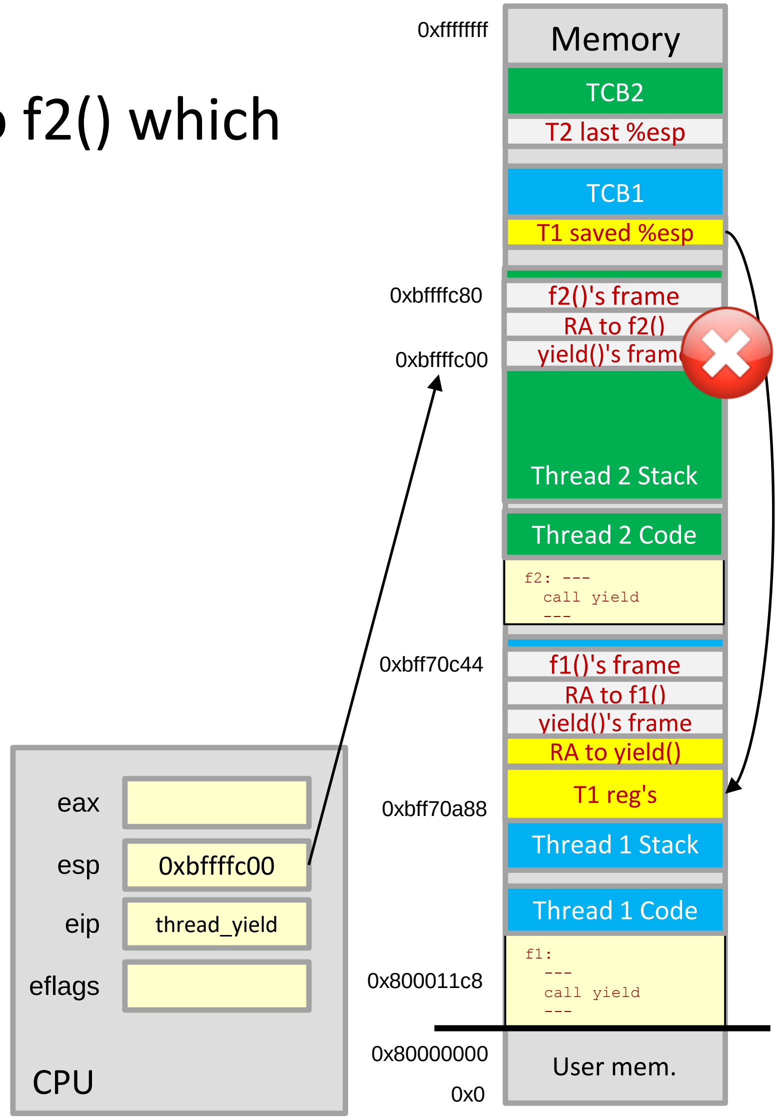
- thread_switch() completes by popping/restoring the registers from T2's stack
- thread_switch() then returns back in the context of thread 2 and not thread 1 (which called it)

```
thread_switch:
   ...

   # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
   mov thread_stack_ofs, %edx

   # Save current stack pointer to old thread's stack
   movl SWITCH_CUR(%esp), %eax
   movl %esp, (%eax,%edx,1)
   # Restore stack pointer from new thread's stack.
   movl SWITCH_NEXT(%esp), %ecx
   movl (%ecx,%edx,1), %esp

   # Restore caller's register state.
   popl %edi
   popl %esi
   popl %ebp
   popl %ebx
   ret
```



Thread Context Switch Example

- thread_yield will then return back to f2() which resumes execution



In-Depth

THREAD CREATION

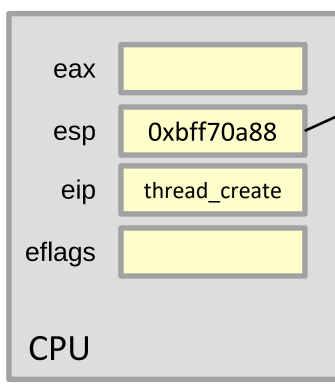
Idea for Creation Mechanism

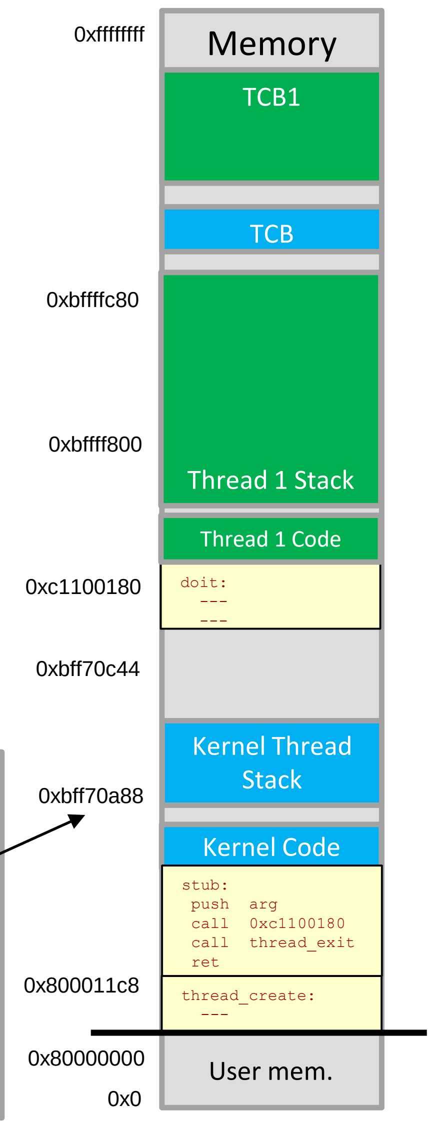
- Allocate a TCB and stack
- Setup the stack to look exactly as if the new thread was already alive and had just called yield()
  - Meaning: Setup the initial stack with dummy "saved" register values and a return address already on it that can be popped by thread_switch()

Thread Create Example

- Assume a new thread should be created with entry point of doit(void* arg)
  - OS will provide a stub function that will call the entry point of the new thread once it is ready
- To create a new thread the kernel will execute thread_create()
- thread_create() will allocate a new TCB for the thread (TCB1) and memory for its stack

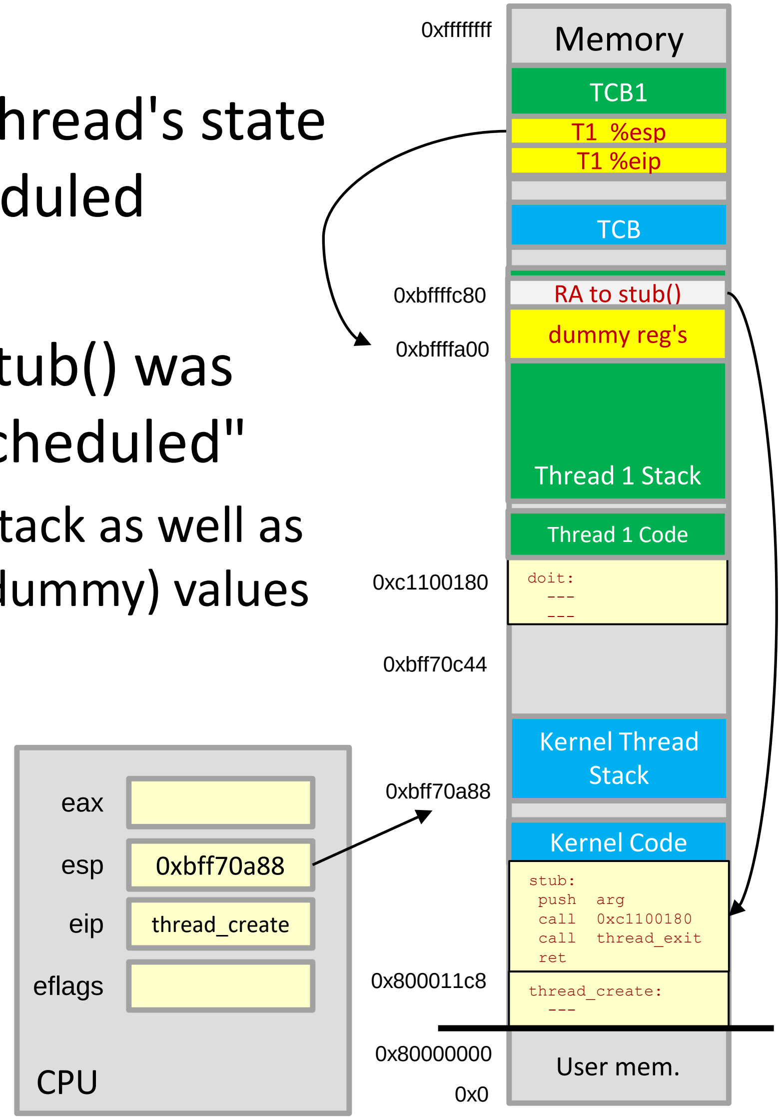
```
void stub( void (*func)(void*), void* arg)
{
  (*func)(arg);
  thread_exit(0);
}
```





Thread Create Example

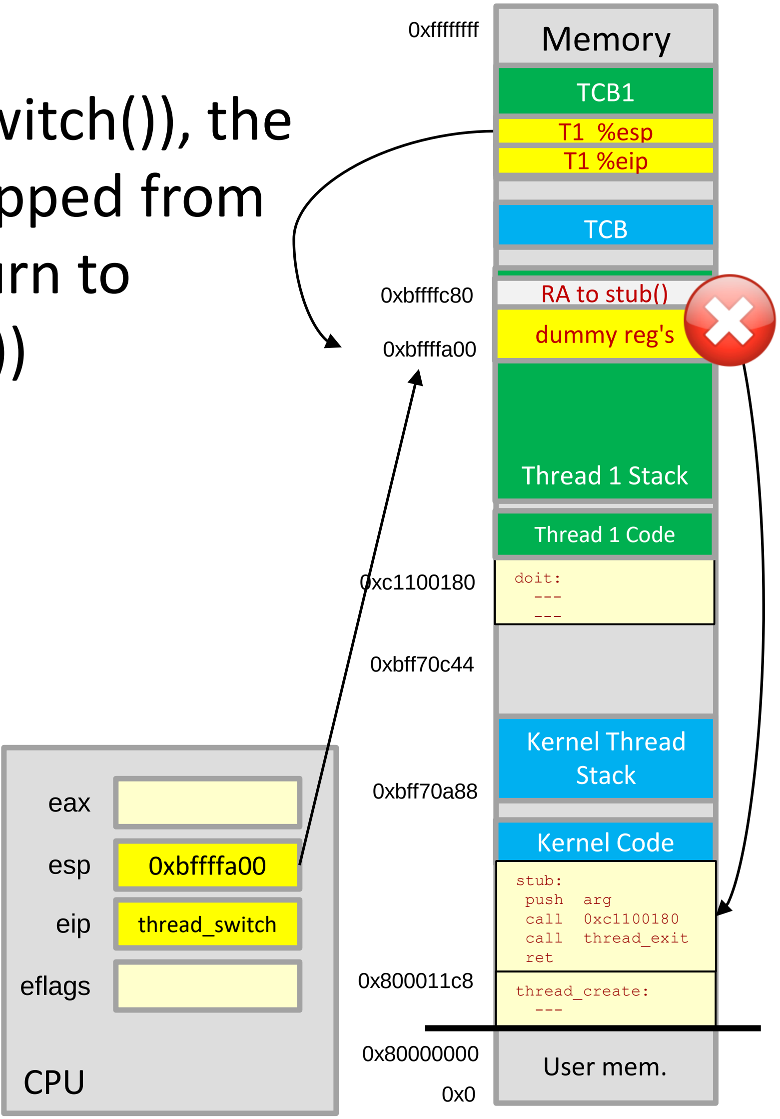
- thread_create() will setup the new thread's state to exactly resemble that of a descheduled (waiting) thread
- To do this, it first makes it look like stub() was the caller when the thread got "descheduled"
  - Pushes the "RA" to stub onto the new stack as well as space representing the "saved" (really dummy) values of the registers
  - Sets the TCB's saved %esp to point at the top of this stack
- Adds this new thread to the ready list to be scheduled on a context switch



USC Viterbi School of Engineering

Thread Create Example

- On a context switch (recall thread_switch()), the dummy registers will be restored/popped from the stack and thread_switch will return to wherever the RA indicates (i.e. stub())

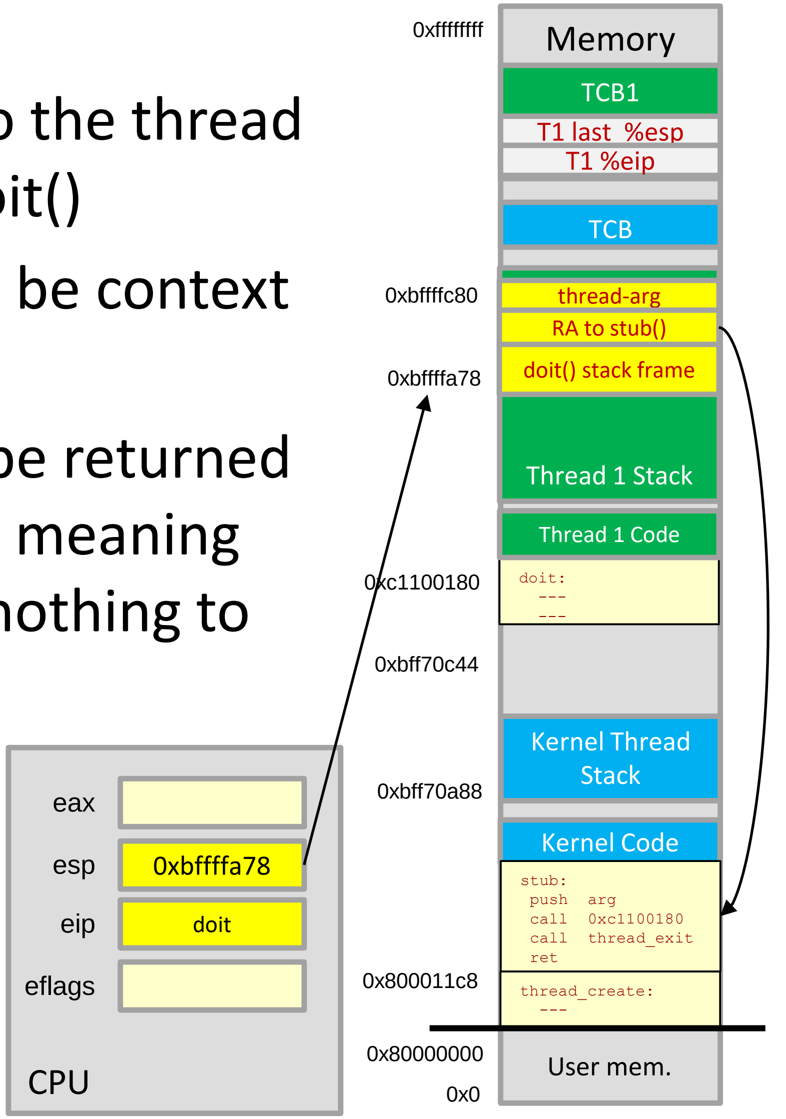



```
void stub( void (*func)(void*), void* arg)
{
  (*func)(arg);
  thread_exit(0);
}
```

Thread Create Example

- stub() will now push the argument to the thread entry point (i.e. doit(arg)) and call doit()
- The thread is now executing and can be context switched as needed
- When doit() completes, control will be returned to stub() which will call thread_exit() meaning stub() will not return (since there is nothing to return to)

```
void stub( void (*func)(void*), void* arg)
{
  (*func)(arg);
  thread_exit(0);
}
```



KERNEL VS. USER THREADS

General Relationship of Threads in User and Kernel Mode

- Each user level thread may have it's own kernel stack for use during interrupts and system calls
- Due to the overhead of a system call and switching from user to kernel mode, some older systems have user-level threads
  - 1 kernel thread
  - Many user threads that the user process code sets up and swaps between
  - User process uses "signals" (up-calls) to be notified when a time quantum has passed and then swaps user threads