Process Structure, Switching

```
process execution states:
  diagram: user/kernel, process mem, kernel thread, kern stack, pagetable
  process might be executing in user space
    with cr3 pointing to its page table
    user mode, so can use PTE_U PTEs, < 0x80000000
  or might be in a system call in the kernel
    e.g. open() finding a file on disk
    process's "kernel thread"
    kernel mode, so can use non-PTE_U PTEs
    using kernel stack
  or not currently executing

xv6 has two kinds of transitions
  trap + return: user->kernel, kernel->user
    system calls, interrupts, divide-by-zero, &c
    hw+sw saves user registers on process's kernel stack
    save user process state ... run in kernel ... restore state
  process switch: between kernel threads
    one process is waiting for input, run another
      or time-slicing between compute-bound processes
    save p1's kernel-thread state ... restore p2's kernel-thread state

Q: why per-process kernel stack?
   what would go wrong if syscall used a single global stack?

how system call works:
   user process uses software interrupt instruction
     int $0x80
   trapframe pushed on top of kstack
     partly by hardware (eip, cs, eflags, esp, ss)
     rest by software (first few instructions in handler are push instructions)
     handler calls the appropriate functions (e.g., syscall functions)
   syscall arguments passed from user to kernel in registers
     kernel function can access user register values in trapframe
   syscall return value passed in register (eax)
     kernel function overwrites trapframe->eax with the return value
   on syscall return (through iret), the trapframe is popped from kstack
     partly by software (the last few instructions before iret are pops)
     partly by hardware (iret pops eip, cs, eflags, esp, ss)

Discuss how syscall arguments and return values can be passed through pointers
  as the kernel and the user process are living in the same address space. This
  makes for very efficient communication between user and kernel, and justifies
  the 2GB of virtual address space that the kernel eats up from the
  process.

In an alternate organization, the kernel could have lived in a different
  address space, in which case, communication would have involved copying
  data from one address space to another. This has the advantage of
  strong isolation between different components (E.g., user/kernel), but
  are less efficient. "Microkernels" take this approach, and are used in
  safety-critical applications, like embedded systems.

if process calls "yield" syscall or if an external timer interrupt is received:
   if executing in user mode (definitely true for yield)
      hardware switches to kernel stack (per process) and pushes eip, cs, eflags, esp, ss at top
      first few instructions push of handler push the rest of user registers
      handler calls the appropriate functions (e.g., scheduler)
      scheduler decides if the current process should continue running?
      if so
          it simply returns from the scheduler
          eventually, returning to the user mode by popping the trapframe from kstack
          trapframe popped partly by software and partly by hardware (just as before)
      if not
         do context-switch:
         the scheduler switches to new process's kstack (which needs to run next)
         old process's kstack contains all information about its trapframe,
              and its callchain from handler to scheduler
         the old process kstack is linked through the its pcb (process control block)
         the saved kstacks are in a state such that switching to them starts running
              the process exactly from where it was preempted
              -  this means that they contain the eip value from where to resume (in kernel)
              -  they also contain the values of other registers, so execution can resume
                 as though it was never interrupted.
              -  typically, on resumption of the execution, the new process will return
                 from the scheduler, eventually returning to user mode by popping the
                 (saved) trapframe from kstack
              -  trapframe popped partly by software and partly by hardware (just as before)
```

```
   if executing in kernel mode
      hardware does not need to switch stacks. it simply pushes eip, cs, eflags
      handler saves other registers and calls scheduler (as  before)
      scheduler either returns or context-switches.
      if it returns, it simply resumes execution in kernel mode from where it
         was preempted
      it it context-switches, it saves kstack, so that future scheduling of
         this process can resume from where it was preempted (by returning from
         the scheduler, as though it was never switched-out).


Thus, a saved kstack contains a trapframe at its top
somewhere it also contains a "context" frame, which is used
    to save resume kernel-mode execution
A kstack may contain more than one trapframes
  - for example, if a timer interrupt was received in the middle of kernel execution

kernel stack diagram (saved):
  top ->
                esp, ss
                eip, cs
                ...
                gs fs es ds
  p->tf ->      edi & 7 other registers
                ---

                ... (information about function call chain inside kernel)
                ... (e.g., return address, arguments, local vars, etc)

                ---
                eip
                ebp
                ebx
                esi
  p->context -> edi
                ---

                ...
  p->kstack ->  ...
  (bottom)
```

Overheads of Timer Interrupt Handling

```
Let's say, you have written a fancy program that really cares about
its speed. Also, assume that when this program is running on the system,
it is the only program on the system. So, whenever a timer interrupt is
received, execution switches to kernel mode, where the scheduler is called,
only to realize that it is the only process that is running, and the
scheduler returns without switching the kernel stack (i.e., no context
switch). So, you are naturally worried about the unnecessary overhead
of handling the timer interrupt periodically, because it serves no
purpose (it merely returns to executing the same process).

Let's look at the overhead of timer interrupt handling:

Typical timer interrupt frequencies: 10-100 Hz (i.e., every 10-100ms)

Approximate time to execute one instruction: 1-100 nanosecond (1-100ns)

Approximate number of instructions executed in one timer interval:
    10ms/1ns = 10ms/10ns = 10^6

Approximate number of instructions executed for handling the timer interrupt:
    say 100-1000 (conservatively 10^3)

Conservative overhead: 10^3/10^6 = 0.1%

Hence, the overhead is insignificant. In fact, it is possible to further
increase timer frequency (to say 1000Hz) without incurring noticeable
overheads. However, as we will discuss later in our discussions on
scheduling, timer frequency beyond a certain value is not very useful.
```