Fork, Process kstack, Context Switching

Process kstack

```
How to determine the kstack size?
  - look at max function call depth in your code
      corollary: do not use deep function call chains
  - look at max size of local variables in functions
      corollary: do not allocate large variables on stack. allocate
                 them on heap if needed.
  - look at size of trapframe, etc.

xv6 uses a 4KB kstack (per process). Linux uses 8KB kstacks.

BUT: we said that any trap causes a trapframe to get pushed
on the kstack. So if there are many external interrupts, one after
another, could the kstack overflow?
Solution:
- Ensure that the kernel cannot cause any software interrupt or
  exception while executing a handler.
- Ensure that handlers of external interrupts (e.g., timer, disk, network, etc.)
  always execute with interrupts disabled.

This ensures that there can be at most two trapframes on the kstack: the first
due to a software interrupt/exception, and the second due to an external
interrupt received while executing in the kernel. This allows you to calculate
an upper-bound on the stack size.

What happens if an interrupt is received while the CPU was executing with
disabled interrupts? The interrupt is ignored:
  - Typically, the interrupting hardware expects an acknowledgement from
    the CPU that it received the interrupt. If it does not receive an
    acknowledgement, it retries the interrupt
  - Also, the CPU has a buffer (of size 2 say) where it stores pending
    interrupts (i.e., interrupts that were received but not serviced).
    As soon as the CPU re-enables interrupts, the pending interrupts are
    delivered to it.

How does an OS keep time? One common way is to count the number of timer
interrupts received.
```

Fork

```
how fork system call works:
  the fork function creates a new PCB and a new kstack (child's)
  it copies the trapframe from the current kstack to the child's
  it changes the return value (eax) in the trapframe for the child
  it also initializes a few more entries on the child's kstack
     so that it can control what are the first few instructions that run
     when the child gets scheduled.
  it adds the PCB to the list of schedulable processes, and returns.


There is one kstack per CPU, which is used by the scheduler thread (the
  first thread to run on that CPU). This kstack is not associated with
  any process. Let's call this the scheduler's kstack.

At bootup, the CPU initializes itself to run with its scheduler kstack.
   Now it will allocate the first process (including its kstack), initialize
   it, and switch from the scheduler's kstack to the new process's kstack.
   The new process will get to run, and will likely call exec/fork to
   create more processes.

Each time a process wants to yield (either voluntarily or preemptively),
   it switches to the scheduler's kstack. The scheduler then finds a process
   to run, and switches to its kstack.

The swtch function on sheet 27 switches kstacks. Let's look at it.
   It saves the current registers on stack
   It then saves the current esp into its first argument (struct context **old)
   It loads the new esp from its second argument (struct context *new)

   The kstack of every suspended process looks like it has just been
     switched out from inside the swtch function.

   Similarly, when a process is running on the CPU, the kstack of the
     scheduler (stored in global variable cpu->scheduler) also looks like
```

```
     it has just been switched out using the swtch function.

swtch sheet 26
  save current thread's registers and stack
  load new threads stack and registers
  saves current registers on current stack, struct context, sheet 20
  expects new thread's stack to have registers in that format
  stack diagram:
    eip *****
    ebp
    ebx
    esi
    edi
  Q: why these registers?
    callee saved, might have caller's live variables
  same format as struct context
```


```
swtch sheet 26
  save current thread's registers and stack
  load new threads stack and registers
  saves current registers on current stack, struct context, sheet 20
  expects new thread's stack to have registers in that format
  stack diagram:
    eip *****
    ebp
    ebx
    esi
    edi
  Q: why these registers?
    callee saved, might have caller's live variables
  same format as struct context
```


Process structure

```
how does xv6 store process state?
  struct proc sheet 20
  kernel proc[] table has an entry for each process
  discuss pgdir, kstack, state, pid fields.
  then discuss ofile field (fd table)

discuss alltraps and trapret on sheet 30
  in the course of a regular trap, trapret gets pushed to stack
     by the call instruction.
  in the case of a newly forked process (or the first process), the
     stack is initialized to contain trapret just below the trap
     frame.
  Why do we save all registers (and not just callee-saved regs) here?
```