Processes in xv6 switch from one user process to another via context switching, which is driven by timer interrupts. This consists of a user-kernel transition such as a system call or interrupt to the old process's kernel; a context switch to a new process's kernel thread, and a trap to return to the user-level process. The use of two context switches helps to simplify cleaning up user processes, as the scheduler runs on its own stack and scheduler thread.

Every process has its own kernel stack and register set, called a `context`. When switching from one thread to another, a process's kernel thread calls `swtch`, which saves the old thread's CPU registers and restores the previously-saved registers of the new thread. This is done by pushing the current CPU register onto the kernel stack and saving the stack pointer to `context *old`. The stack pointer is then copied to `*new` and the previously saved registers are popped from the stack. Both the stacks and code being executed will then be switched as expected.

Since xv6 consists of multiple CPU's, measures must be taken to avoid race conditions. Thus, a process that wishes to give up the CPU must acquire the *process table lock*. It then releases any other locks, updates its state, and then calls the scheduler to switch between processes. `Sched` checks that `yield`, `sleep`, and `exit` have been called, and that interrupts are disabled while a lock is in place. It then calls `swtch`, which, as mentioned before, saves the current context and switches to the scheduler's context. `Sched` is also responsible for unlocking the process table after a process has run and stopped, and another is ready.

When one wants a process to wait until some condition is met, a `sleep` call will be invoked. It acquires the process table lock to ensure that the `sleep` and `wakeup` calls do not miss each other, and then sits in a conditional loop asleep.

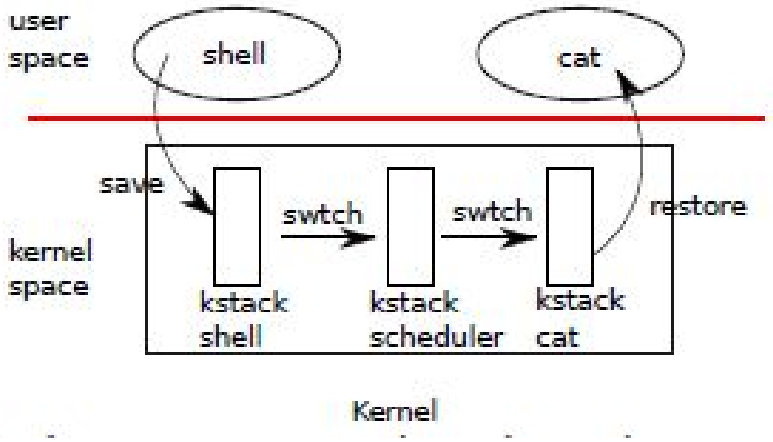

Figure 1. Context Switching Diagram