Read chapter-5 for today’s lecture.

We have discussed locking for uniprocessor hardware. Let us quickly review our existing understanding. A lock is used to restrict the concurrent execution of a block of code (also called the critical section). A locking framework has two interfaces, acquire and release. The piece of code between "acquire" and release is called the crucial section. There can be several implementations of "acquire" and release. e.g., acquire and release can be implemented by simply disabling and enabling the interrupts on uniprocessor hardware. Alternatively, semaphores can also be used to implement acquire and release. We will study in future lectures that several other implementations are also possible. Right, now you can assume the implementation of "acquire" and release as black box. The only thing which is important for today’s discussion is: two threads can never execute simultaneously in a critical section.

Most of the locking interfaces also take a lock variable as an input. You can think of a lock variable is a variable which is attached to a critical section. For example, on this slide, critical_section1 and critical_section2 are protected by different locks, and hence they can execute in parallel.

Let us talk about scheduling. In xv6, the schedule is called explicitly when a process is waiting for an event, or I/O, or going to sleep. Apart from these events, a timer interrupt may schedule a new process when the process has completed its time slice. You might wonder why the xv6 schedules process instead of a thread. This is because in xv6 all processes are single threaded.

One of the design decisions, xv6 has to make is when to do a context switch and how to handle concurrent calls to the scheduler.

Let us look at Figure 5-1 in the xv6 book. Every process has its own kernel stack, and the scheduler runs on its own kernel stack. When a CPU yields, the context of the current process is saved, and the context of the scheduler is loaded. The scheduler then picks a runnable process and do a context switch to load the context of the target process.

Scheduler maintains a list of all processes (PCBs). PCB contains a pointer to the process context.

During the context switch, the process allocates space for saving its context, save the registers in the allocated space and save the pointer of the process context in the PCB. A context switch is done by the swtch:2958. swtch allocates space for process context on the stack. Due, to calling convention caller-saved registers (eax,edx,ecx) are dead at this point, swtch only allocate space for callee-saved registers. EIP is already saved on the stack by the caller, so swtch do not need to save them explicitly. After, allocating space for context the current stack pointer is the address of the context itself. Due, to this reason swtch does not explicitly, saves stack pointer because the address of context (effectively stack pointer) is saved in the current process's PCB by the swtch routine. After, saving the context the swtch function load the context of the target process in the stack pointer and restore the context by popping them from the stack.

Read chapter-5 for the rest of the lecture.