1) Operating systems like Windows, Linux, and Solaris use multiple locking mechanisms because different situations need different solutions. Spinlocks work best when you only need to lock something for a really short time, like a few microseconds. The thread just keeps checking over and over if the lock is free, which sounds wasteful but actually saves time because switching between threads takes even longer. Mutex locks are better when the lock might be held for a while because they put the waiting thread to sleep instead of making it waste CPU power just sitting there checking. Semaphores are more flexible since they can let multiple threads access something at once and any thread can signal them, not just the one holding the lock. This makes them good for things like producer-consumer problems. Adaptive mutex locks try to get the best of both worlds by spinning for a bit first and then going to sleep if the lock still isn't free. Condition variables let threads wait for specific conditions to happen, not just for a lock to open up, and they work together with mutexes. Basically, operating systems give us all these options because what works great in one situation might be terrible in another - you wouldn't want to spin for 10 milliseconds wasting CPU, but you also wouldn't want to put a thread to sleep for something that only takes 10 nanoseconds.

2) Spinlocks don't make sense on single-processor systems because there's a fundamental problem when one thread is spinning and waiting for a lock, it's using up the whole CPU doing basically nothing. Meanwhile, the thread that actually has the lock can't even run to release it because the spinning thread is hogging the only processor available. So thread B just wastes its entire time spinning while thread A sits there unable to run, which is completely pointless. But on multiprocessor systems, spinlocks actually work really well because the thread holding the lock could be running on a different processor right at that moment. If the lock is only held for a tiny amount of time, it makes sense to just spin and wait because switching threads is expensive , you have to save everything about the current thread, pick a new one, load all its stuff, and then do it all again later. For really short waits, that whole process takes longer than just spinning would. That's why spinlocks are useful in kernel code and interrupt handlers on multiprocessor systems where things happen super fast.

3) The system clock works by using timer interrupts that go off at regular times to keep track of what time it is and handle scheduling stuff. When you disable interrupts, those timer interrupts can't do their job, so the system basically loses track of time. Like if interrupts are off for five milliseconds and the timer fires every millisecond, the system misses five ticks and falls behind. This gets worse when interrupts are turned off a lot because all those missed ticks add up and the clock drifts away from the real time. This messes up programs that need accurate timing and also breaks the scheduler since it can't interrupt processes anymore. To avoid these problems, you should only disable interrupts for super short periods, like a few microseconds. Modern systems try to use other synchronization methods like spinlocks or mutexes that don't need to disable all interrupts. Some systems can even disable just certain interrupts while leaving the timer ones running. The best approach is to not disable interrupts at all if you can help it and just use better locking methods instead.