Lab 2

3.2

In general, when a backend hangs due to *panic()* being called, *clktime* will not be updated. This is because *clktime* is incremented in *clkhandler.c*, which is invoked by *clkdisp.S*. *clkdisp.S* is called in *clkinit.c* through the *set_evec()* function, which sets the interrupt vector for the clock to invoke *clkdisp()*. This means that *clktime* is updated through the clock interrupt with a 1 millisecond interrupt rate. Hence, in *panic.c*, *disable()*, which disable all interrupts, is called before the while loop causes the backend to hang. This means that the clock interrupt is never enabled again, as *panic()* causes the backend to hang, which means that *clktime* is never updated again.

3.3

When interrupt 0 and interrupt 6 are generated using inline assembly, the lower half XINU code (??) is only executed once instead of repeatedly. We can verify this by checking that the final value of *divzerocount* is 1 in *main.c* and the final value of *invalidopcount* is 1 in *evec.c*.

4.2 + Bonus

To verify that my trapped system call implementation works correctly, I tested my wrapper on its own before testing that both the wrapper and dispatcher work together. For the former test, I would call “int $34” and call the kernel function from _Xint34. Additionally, in _Xint34, I made sure to push any passed-in arguments onto the stack and clear them before returning. Next, I verify that the EAX register successfully passes the value to the variable to be returned from the kernel function.

To test both the dispatcher and wrapper, I switched back to “int $33” in the wrapper function and assigned the returned value to a *syscall* variable. For example, to test my implementation for *listancestorsx()* with an argument of 3, I assigned the return value to *numancestors*, which is of type *syscall*. Then, I printed out the value of *numancestors* to check that it was 2. This is repeated for the other four wrapper functions.

The code used for testing in *main.c* is as below:

```c
/* Test for 4.2: Software layers*/
syscall pid = getpidx();
kprintf("in main, expected: 3, pid = %d\n",  pid);

syscall numancestors = listancestorsx(pid);
kprintf("in main, expected: 2, numancestors = %d\n",  numancestors);

pid32 p1 = create(foo, 1024, 20, "proc1", 0, NULL);

syscall oldprio = chpriox(p1, 25);
kprintf("in main, expected: 20, oldprio: %d\n", oldprio);

pid32 p2 = create(foo, 1024, 30, "proc2", 0, NULL);

syscall prio = getpriox(p2);
kprintf("in main, expected: 30, prio: %d\n", prio);

pid32 p3 = create(foo, 1024, 20, "proc3", 0, NULL);

pid32 p4 = create(foo, 1024, 20, "proc3", 0, NULL);

int i;
for (i = 0; i <= p4; i++) {
    struct  procent *prptr = &proctab[i];
    kprintf("pid: %d, process state: %d\n", i, prptr->prstate);
}

syscall numproc = procrangex(0, 0 , 10);
kprintf("in main, numproc = %d\n",  numproc);
```