ECE 3849 Homework 2

Ezra Davis

11/10/2016

Problem 1

A)

The `++iHead;` line is a read-modify-write operation, and if the high priority interrupt is called in the middle of this operation, then the low priority task will continue to write it's iHead value, ignoring the change in the high priority task.

B)

The high priority task doesn't need to be changed - it won't be interrupted.

```
void SinkTask (void)  // low priority
{
    int iValue;
    while (TRUE)
        IntMasterDisable();     ///// Line changed //////
        if (iHead != iTail)
        {
            iValue = iQueue[iHead];
            ++iHead;
            if (iHead == 100)
                iHead = 0;
            IntMasterEnable(); ///// Line changed //////
            <do something with iValue>;
        } else {               ///// Line changed //////
            IntMasterEnable(); ///// Line changed //////
        }                      ///// Line changed //////
}
```

Problem 2

Line 4 is by far the most indicative of the fact that the function is not reentrant. It is a non-atomic (read-modify-write) operation on a global variable, which means that if the code is interrupted after the read, but before the modify, the changes made to `iCount` during the higher priority task will be ignored.

Also, line 5 will be not reentrant depending on the implementation of `printf`, which usually uses global variables and is not reentrant.

Problem 3

```
static int iValue;
int iFixValue (int iParm)
{
    int iTemp;
    IntMasterDisable(); ///// New line /////
    iTemp = iValue  // READ
    iTemp += iParm * 17;
    if (iTemp > 4922)
        iTemp = iParm;
    iValue = iTemp; //WRITE
    IntMasterEnable(); ///// New line /////
    iParm = iTemp + 179;
    if (iParm < 2000)
        return 1;
    else
        return 0;
}
```

Problem 4

Step	Task1	Task2	semTask1	semTask2	sem0
1	Running	Ready	0, none	1, none	1, none
2	Blocked	Ready	0, Task1	1, none	1, none
3	Blocked	Running	0, Task1	0, none	1, none

Notes to self (feel free to ignore):

At time 2, `Task2` is running and *might be* in the critical section, but is definitely before `Hwi0` runs - because then `Task1` wouldn't be blocked, and would be running instead of `Task2`. This means that `sem0` could actually be in either state... but because of the location that time 2 is indicated, `Task2` probably hasn't been running long enough to enter the critical section.

After time 2, the `Hwi0` runs, interrupting `Task2` while it's running. `Task1` runs until it reaches the line where it `pends` on `sem0`, after which `Task2` runs again.

Once `Task2` posts to `sem0`, control switches over to `Task1` again, which runs its critical section, but stops when it `pends` on `semTask1`. Then `Task2` runs until it reaches the top of the while loop (probably just one `jmp` instruction) and we reach time 3.

Timeline:

- `Task1` starts running.
- Time 1 happens
- `Task1` Runs briefly until it reaches `Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);`, when it is blocked.
- Time 2 happens somewhere in here - probably immediately before `Task2` starts running, but technically, it could be after...
- `Task2` starts running, and after decrementing `sem0`, `Hwi0` happens, and switches control over to `Task1`
- `Task1` runs until it `pends` on `sem0`, after which it is blocked.
- `Task2` starts running until it `posts` to `sem0`
- `Task1` suddenly takes over, runs its critical section, and runs until it is blocked by `semTask1`
- `Task2` runs until Time 3 happens, and then it blocks pending on `semTask2`