ECE459 Final W2014

Question 1

1. The read() and write() functions are blocking.

2. http://www.kegel.com/dkftpbench/nonblocking.html

```
void sendFile(const char *filename, int socket, sendfile_state c) {
        int nread, nwrite, i;

        /* Force the network socket into nonblocking mode */
        setNonblocking(socket);

        c.socket = socket;
        c.fd = open(filename, O_RDONLY);
        c.buf_len = 0;
        c.buf_used = 0;
        c.state = SENDING;
}
int handle_io(sendfile_state c) {
        if (c.state == IDLE) return 2;
        if (c.buf_len == c.buf_used) {
                c.buf_len = read(c.fd, c.buf, BUFSIZE);
                if (buf_len == 0) {
                        close(c.fd); close(c.socket);
                        c.state = IDLE;
                        return 1;
                }
                c.buf_used = 0;
        }

        nwrite = write(c.socket, c.buf + c.buf_used, c.buf_len - c.buf_used);
        c.buf_used += n_write;
        return 0;
}
main() {
        sendfile_state c;
        int sock = fileno(stdout);
        sendFile("foo.txt", sock, c);
        do {

            int done = handle_io(c);
            if (done) break;

        }

    }
```

Question 2

1. work():
   For the first loop, every iteration depends on results from the previous iteration. For the second loop, loop iterations do not depend on each other. For the third loop, loop iterations do not depend on each other.

   slow():
   For the first loop, loop iterations do not depend on each other. For the second loop, iterations create side effects, so each iteration depends on results from the previous.

2. Not covered.

Question 3

Not covered.

Question 4

Everything in CUDA instead of OpenCL.

a) read-only: crcTable
   write-only: buffer

b) (8,8,8) blocks, (8,8,8) threads/block

c) Each thread works on its own subset of passwords, determined by the thread ID. The atomic_table_write ensures writes to the result table are atomic.

```
__device__ uint16_t calculate_crc(const char *message, const uint16_t
*crcTable) {
        uint8_data;
        uint16_t remainder = 0;

        for (int byte = 0; byte < 6; ++byte) {
                data = message[byte] ^ (remainder >> (WIDTH-8));
                remainder = crcTable[data] ^ (remainder << 8);
        }
        return remainder;
}
__global__ void kernel(const uint16_t *crcTable, char *buffer) {
        int x = blockIdx.x*blockDim.x + threadIdx.x;
        int y = blockIdx.y*blockDim.y + threadIdx.y;
        int z = blockIdx.z*blockDim.z + threadIdx.z;

        if (x >= 62 || y >= 62 || z >= 62) return;

        char password[6];
        password[0] = decode_char(x);
        password[1] = decode_char(y);
        password[2] = decode_char(z);

        for (int i = 0; i < 62; ++i) {
                password[3] = decode_char(x);
                for (int j = 0; j < 62; ++j) {
                        password[4] = decode_char(y);
                        for (int k = 0; k < 62; ++k) {
                            password[5] = decode_char(z);
                            uint16_t crc = calculate_crc(password, crcTable);
                            atomic_table_write(buffer, crc, password[0 … 5]);
                        }
                    }
                }
            }
```

Question 5

1. We can store the result from the first getFifth() call and use it in the second print statement.

2. The existence of another thread that modifies the linked list would make the transformation unsafe, as the result of getFifth could change in between the two printf statements.

3. In this scenario, the transformation would be safe, as there are no other threads that can modify the list, and no interrupts. getFifth does not modify the list structure and has no side effects. The compiler would likely not perform this optimization, as it may not be able to identify that getFifth has no side effects or that the linked list is not modified, partially due to array aliasing.

Question 6

1. a=3, b=1, c=2;   a=2,b=3,c=1

2.

Thread 1	Thread 2	a	b	c
int t = *x		1	2	3
	int t = *x	1	2	3

*x = *y		1	1	3
*y = *t		2	1	3
	*x = *y	2	3	3
	*y = *t	**2**	**3**	**2**

3. Not covered. Adding restrict tells the compiler that for the lifetime of the pointer, no other pointer will be used to access the object to which it points. This does not make the function thread-safe.

4. lock(m) at beginning of function and unlock(m) at end. Ensures that other threads cannot read or write to x and y in the middle of a swap.