1 Objective

The purpose of this programming assignment is to illuminate the principles of speeding up data-touching protocol operations, by having you apply them to a simple example protocol. You will be given a “reference implementation” of a (portion of a) simple protocol that involves reading and writing user data bytes. Your job is to improve the performance of the implementation while maintaining its correctness according to the specification. You are to do this by applying the principles discussed in class, as well as any other “tricks” you know or can find. Your success will be measured by how much “speedup” you obtain on a test data series, compared with the reference implementation.

2 Protocol Specification

The protocol you will be dealing with this time can be viewed as an application-layer protocol, in that the implementation is incorporated into the application code, in user space. The protocol provides two functions: encryption using a stream cipher, and error-detection using the Fletcher checksum.

You will need to modify only the output routine of the protocol. The function takes a payload (contained in a BUF chain, as in the previous assignment) and an auxiliary function as input. It returns a BUF chain containing the same payload, encrypted with the stream cipher, and with two checksum bytes appended; the checksum is over the encrypted data. The relevant type signatures for this assignment are:

```c
typedef /* given elsewhere... */ BUF;
typedef unsigned char (*padfn)();
BUF proto_output(BUF payload, padfn f);
```

Your job is to modify `proto_output()` to make it run faster. You may not modify any of the other code for this assignment.

2.1 Encryption

Encryption with a stream cipher is done by XOR-ing each byte of the payload with a byte of the keystream. The `proto_output()` function accesses the keystream via an auxiliary function that is passed as an input parameter. The auxiliary function returns the next byte of the keystream, which can then be XOR-ed with the next byte of the payload. If \( D[i] \) denotes the \( i \)th byte of the payload, the encryption operation applied to each byte is:

\[
D[i] \oplus f();
\]

You will be given a copy of the `nextbyte()` function, which is passed as an argument to `proto_output`, but you may not modify it in any way (that particular function may not be the one used when your implementation is tested).
Note that the length of the payload is not modified by the encryption operation. The stream cipher functions are provided in a file cipher.c, so you can compile them with your code for testing purposes, but you may not modify them in any way.

2.2 Checksum

Computing the Fletcher checksum involves computing two 8-bit values and appending them to the message. Let \( L \) be the length of the message (including the two bytes to be appended, which are initialized to zero), and let \( D[i] \) be the \( i \)th byte, where bytes are numbered from 1 to \( L \). To add the checksum to the outbound message, we first compute:

\[
\begin{align*}
    f_0 &= \sum_{i=1}^{L} D[i](mod 255) \\
    f_1 &= \sum_{i=1}^{L} (L - i + 1)D[i](mod 255)
\end{align*}
\]

Having done this, we compute \( c_0 \) and \( c_1 \) as a function of \( f_0 \) and \( f_1 \), in such a way that when we replace the two final zero bytes with \( c_0 \) and \( c_1 \), and the receiver computes \( f_0 \) and \( f_1 \), both are zero. It can be shown that setting \( c_0 = f_0 - f_1 \), and \( c_1 = c_0 - f_0 \) achieves the desired result.

Here is an example to show how it’s done. Suppose the original message is just three bytes: 1, 2, 3. In this case we use \( L = 5 \), because the final message length will be 5. We have:

\[
\begin{align*}
    f_0 &= 1 + 2 + 3 = 6 \\
    f_1 &= 5(1) + 4(2) + 3(3) + 2(0) + 1(0) = 22 \\
    c_0 &= f_0 - f_1 = 6 - 22 = -16 \mod 255 \\
    c_0 &= 239 \\
    c_1 &= -c_0 - f_0 = 16 - 6 \mod 255 = 10
\end{align*}
\]

So the final message is 1,2,3,239,10. Checking the message at the receiver, we get:

\[
\begin{align*}
    f_0 &= 1 + 2 + 3 + 239 + 10 = 255 \mod 255 = 0 \\
    f_1 &= 5(1) + 4(2) + 3(3) + 2(239) + 10 = 510 = 0 \mod 255
\end{align*}
\]

Note that the information returned by \texttt{proto\_output} is two bytes longer than the input.

2.3 Implementation Specification

The input to \texttt{proto\_output} is a \texttt{BUF} chain containing a single payload. The chain may contain empty buffers; the code should deal correctly with an empty payload. The output should also be a single \texttt{BUF} chain; \texttt{proto\_output} is allowed to rearrange the data, but when processed by the receiver, the result should be the same as the input to \texttt{proto\_output}.

3 Rules of the Game

You may not modify the type signature of \texttt{proto\_output}, nor any of its arguments, nor any aspect of the cipher implementation (the cipher implementation used for evaluation purposes may be completely different). You may modify the body of \texttt{proto\_output} in any way you wish, so long as you do not change its external behavior. (Thus,
P3-5 and P9 are illegal for this assignment.) That is, for any given input, the output of your implementation must be equivalent to that of the reference implementation—in other words, a buffer chain containing the same sequence of bytes.

Your code will be evaluated by compiling it (with optimization turned on at the default level) and running it on one of the machines in the MultiLab; these are approximately 2.5GHz P4s. You may design your code for that specific processor if you wish. You are permitted (even encouraged) to take advantage of any special features or instructions that machine may have; however, you are responsible for figuring out how to do that.

The performance of your code will be evaluated by measuring the time it requires to process a hundred or so pre-fabricated payloads. Measurements will be taken on several identical runs to verify stability and eliminate random influences as much as possible.

4 Hints

- The reference implementation provides many opportunities for applying P1 “Avoid obvious waste”.
- It would be a good idea to read the Clark and Tennenhouse paper (“Architectural Considerations for a New Generation of Protocols”) and consider applying Integrated Layer Processing techniques.
- It will probably be advantageous to apply one or more forms of P2 “Shift computation in time”.
- A fair amount has been written about optimizing the Fletcher checksum calculation. A little bit of research should pay big dividends here.

5 Turnin

Email me a copy of the file “proto.c”, containing your modified code (only), by the due date. Turn in a neatly-printed listing (preferably 2-up) in class.