CS 740, Fall 1996
Homework Assignment 4

Assigned: Oct. 29
Due: Thurs., Nov. 7, 1:00PM

Policy

You may work in a group of up to 3 people in solving the problems for this assignment. However, you must prepare your own writeup, and you must identify your collaborators in your writeup.

1. Cache Protocol Validation with SMV

The files you need for this part of assignment are available in the directory:

`/afs/cs.cmu.edu/academic/class/15740-f96/public/asst/asst4/cache`

The problems all make use of the model checker SMV. This program has been installed in the class AFS directory, although you may need to compile it for your CPU type. A user's manual for SMV is included in the assignment directory.

The problems in this assignment require running SMV many times, making incremental changes to the files (this is the easiest way to gain an understanding of the program). For each problem, show the fragment of the SMV file you are modifying, and indicate the response produced by SMV, i.e., that the verification succeeded, or describing the counterexample it produced. You *should not* include the detailed trace file. A verbal or pictorial explanation would be more useful.

One useful technique for maintaining multiple versions of an SMV description is to create a single "master" copy containing C-style conditional compilation directives. You then generate the different versions by running the C preprocessor using the appropriate set of command line arguments. For example, the version of `snoopy.smv` shown in the assignment directory was generated from the master version `snoopy.master`. The appropriate incantations are stored in the `Makefile`, including some good parameters for running SMV.

1.1. Exploring the Write Invalidate Protocol

The file `snoopy.smv` is an SMV description of a write-invalidate cache protocol, as was described in class. This protocol is similar to the one described in Section 8.3 of Hennessy and Patterson but is somewhat more complete in its details.

Task 1: Tracing Interesting Behaviors

When SMV is run on the cache model, it simply returns a message stating that the verification succeeded. Often we'd like to be convinced that the protocol has the potential to do interesting things, as well. The best way to do this is to purposely run SMV with an incorrect specification that where the desired behavior is a counterexample.

Show the following behaviors can actually occur. Give the temporal logic formulas you use to generate the behaviors, and describe the sequence generated:

1. All 3 caches get a copy of the same cache line.
2. A cache must write out a dirty block to satisfy its processor's read request.
3. The system cycles indefinitely between a condition where both blocks are owned by individual caches, and then where neither block is owned by a cache.

Generalizing the Bus Design

The current bus design (defined by the module `bus`) uses a quasi-round-robin arbitration scheme. That is, there is a token that circulates around defining the relative priorities of the bus requestors. In a strict round-robin scheme, this token would circulate in a fixed progression. In the current design, the token is circulated nondeterministically. Additional *fairness* constraints require that the token be assigned each possible value infinitely often. This is sufficient to guarantee the fairness of the bus protocol, i.e., the part of the specification given by the lines:

```
-- Instruction not stalled indefinitely
   & AG (!EG c0.stall)
   & AG (!EG c1.stall & !EG c2.stall)
```

It would be preferable to remove all implementation artifacts from the bus protocol, i.e., to prove that the protocol is correct for *any* fair bus design. Thus, we would like to eliminate any form of prioritization in the bus module.

Task 2A: Attempting Nondeterministic Bus Granting

Suppose we eliminate the state variable `token` from the bus module, and simply grant the bus master nondeterministically, e.g.,

```
next(master) := {0, 1, 2, no-one};
```

Show that the *safety* condition of the protocol specification can be violated by creating counterexample traces that do each of the following:

1. A block no longer has any owner.
2. A block has multiple owners.

Describe the interactions between the bus and the caches that cause these two conditions to arise.

Task 2B: Safe Nondeterministic Granting

Fix the protocol so that it is safe by modifying either the bus design, the cache design, or both. Your modifications should still avoid any implementation artifacts, i.e., it should not include any code that constrains how the bus arbiter chooses among the requestors.

Task 2C: Attempting to Add Fairness

With your modified design, consider the fairness constraints

```
FAIRNESS
  master = 0
FAIRNESS
  master = 1
FAIRNESS
  master = 2
```

- Does your protocol satisfy the complete specification with these constraints imposed? [The answer to this may depend on how you modified your design].
- Do these fairness constraints eliminate any system behaviors that are, in fact, perfectly legal? If so, demonstrate it by showing that the protocol, under these fairness constraints, satisfies an overconstrained specification.

Task 2D: Correctly Adding Fairness

Create a modified set of fairness constraints that avoids the problems of our previous attempt. That is, it should allow the liveness condition of the protocol specification to be verified while not eliminating valid system behaviors.

Note: You may find it difficult to write these fairness constraints using temporal logic formulas in terms of the existing signal values. You may find it useful to add auxilliary signals to the bus module (in the `DEFINE` section of the module specification) that do not affect the system behavior, but allow the fairness constraints to be specified more succinctly.

2. TCP with On-the-Fly Data Compression

The files you need for this part of assignment are available in the directory:

`/afs/cs.cmu.edu/academic/class/15740-f96/public/asst/asst4/tcp`

As we discussed in class, today's modems compress and uncompress data on-the-fly, often yielding an improvement of a factor of 2 in effective bandwidth. In this part of the assignment, we will experiment with adding on-the-fly data compression and uncompression to TCP.

We have provided a suite of working programs to help you with this part of the assignment. These programs are broken into three groups.

1. `tcpserver.c` and `tcpclient.c`
2. `ttcp.c`
3. `lzcomp.c, lzuncomp.c, compressor_bits.c, and util.c`

The programs in the first group, `tcpserver.c` and `tcpclient.c`, implement a simple TCP client/server pair. These programs are used as follows. First execute the server program, `tcpserver`. The server will then wait for a message from the client program, `tcpclient`. Next, execute the client program from either the same workstation that the server is running on or from another workstation. When you enter the client program, it will first ask you for the internet address of the server. It will then ask you to type in the message that you want to send to the server. If all goes well, the message will print out on the server's screen, and both the client and server programs will terminate. (Note: the message ends at the first space, tab, or carriage return.) These files are provided only to illustrate how a TCP connection is established, and for testing.

The second group consists of a single program, `ttcp.c`. This program is used to test the performance of TCP. You can read more about it in file `README.ttcp`.

The third group of programs implements the Lempel-Ziv data compression algorithm. The program `lzcomp.c` reads raw data from the standard input stream, and outputs compressed data on the standard output stream. The program `lzuncomp.c` reads compressed data from the standard input stream and outputs uncompressed data on the standard output stream.

A `Makefile` is provided to help you compile these programs.

Task 3

Your task is to incorporate data compression into TCP and to evaluate its performance. In particular, you are to provide a set of subroutines `lzread` and `lzwrite` that replace the standard Unix subroutines `read` and `write`. The new routines should have the same external interface as the old routines, i.e., you should be able to replace the calls to `read` and `write` in `ttcp.c` with calls to `lzread` and `lzwrite` with no changes in the argmuments or return values of these functions. You may find it helpful to reuse code from the programs that implement the Lempel-Ziv compression algorithm.

There is one subtle problem that you will have to solve. When the compressor reads a byte of input data, it may generate fewer than 8 bits of output data. Hence, it will not necessarily have a byte to output. This is not a big problem if the entire input stream is presented to the compressor at once, since only the very last byte of the output stream can be left in the pipe, and it is not difficult to flush this byte out. Your function `lzwrite`, however, will receive the input stream in pieces (however `ttcp` chooses to call it), and you must ensure that at the end of each call, nothing is left in the pipe, so that the external behavior of `lzwrite` and `lzread` does not differ from that of `read` and `write`.

Once you have `lzread` and `lzwrite` working, you should use `ttcp` to compare the performance of compressed TCP connections with uncompressed connections. Try establishing a connection between two ports on the same machine, between two ports on different machines on the same Ethernet, and between machines separated by some distance. You may find that some fine-tuning is required in order to make compression worthwhile.

You should hand in your code electronically (by sending it in email to `bmm@cs.cmu.edu`), and handin a hard-copy writeup of your experimental results.

The remainder of this section explains the Lempel-Ziv data compression algorithm.

Data compression

The purpose of data compression is to take a file (or stream of data) A and, within a reasonable amount of time, transform it into another file B in such a way that (1) B is smaller than A, and (2) it is possible to reconstruct A from B. A program that converts A into B is called a *compressor*, and one that undoes this operation is called an *uncompressor*.

Of course, a compressor cannot guarantee that the file B will always be smaller than A. It's easy to see why: if this were possible, then imagine what would happen if you just kept iterating the method, compressing the output of the compressor. In fact, with a little bit of thought, you should be able to convince yourself that any compressor that compresses some files must also actually enlarge some files too! Nevertheless, compressors tend to work pretty well on the kinds of files that are typically found on computers, and they are widely used in practice.

The Lempel-Ziv Algorithm

The programs that we provide implement a version of the *Lempel-Ziv* data compression algorithm. Lempel-Ziv is an example of an *adaptive* data compression algorithm. What this means is that the code that the algorithm uses to represent a particular sequence of bytes in the input file will be different for different input files, and may even be different if the same sequence appears in more than one place in the input file.

The basic data structure used by the Lempel-Ziv algorithm is a tree. Initially, the tree consists of a single node, the root. As the input file is processed, nodes are added to the tree. Eventually, each node in the tree may have as many 256 different children, one for each possible value of an 8-bit byte. Each node in the tree except for the root holds a byte value (conceptually, you can think of this byte as being on the edge between the node and its parent). Each node is also labeled according to when it was added to the tree. The root is labeled 0, the first node added to the tree is labeled 1, and so on.

The way that you compress a piece of text is as follows. Starting at the root, you read in bytes one by one, using them to trace a path down the tree. For instance, if you read the bytes `'a'`, `'b'`, `'c'`, you would go from the root to its child that has byte `'a'` in it, then to that node's child having byte `'b'`, and then to that node's child having byte `'c'`. (If we think of bytes as being on the edges, we can think of this as the `abc` path from the root.) Once you reach a node for which the desired child doesn't exist (maybe the next character read in was `'b'` and the current node doesn't have a child with byte `'b'`) you do the following. First, you print out the label of the current node. Then you create a new child of the current node and give it the byte value of the character read in as well as the next available label. They you print out that byte value (this is so the uncompressor will know what byte it was) and you go back to the root.

The idea is that the label of a node is representing the sequence of byte values stored on the path from the root to that node. Think about what happens if the file we're compressing has only "a"s in it. We will read in the first `a` and make an associated child of the root. Then we will read in two "a"s and make a new node (the 'a' child of the last node we created). Then we read in three "a"s and make a new node, and so on. Pretty soon, our compressor is able to read in a large number of "a"s in between having to print anything out.

When you reach the end of the input file, the program just outputs the label of the current node.

Uncompressing a File

The algorithm for uncompressing a file is very similar to the algorithm for compressing a file. The uncompression algorithm will build up the exact same tree that the compression algorithm builds up. In addition, the uncompression algorithm will maintain an array of pointers to nodes in the tree. The ith element in this array will be a pointer to the node with label i.

The algorithm begins by reading in the first label in the compressed file. The array is used to determine which node the label refers to. The algorithm prints out the sequence of bytes on the path from the root to this node. It then reads in the next byte in the compressed file and creates a child of the current node with that byte value (mimicking the compressor) and prints out that byte. It then continues on to read the next label and so forth.

You might be wondering whether or why this compression algorithm will compress a file. It is based on the observation that sequences of bytes tend to be repeated over and over again within the same input file. For example, the sequence "compressed" appears many times in this document. Like we saw above, if you see sequences over and over, they will result in long paths in the tree, which means that a large number of characters get replaced by a single node label.

References

You can learn more about the TCP protocol, and about Unix network programming, from the following two books,

1. D. E. Comer, *Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1991.

2. W. R. Stevens, *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

Both are well written. You might be able to complete this assignment without opening either book. If, however, you have questions about the network programming network functions used by the programs that we have provided, turn to Chapter 6 of the book by Stevens.