

Merry Christmas! Linux RULES! New applications to develop!

Source: <http://linux.derkeiler.com/Newsgroups/comp.os.linux.misc/2007-07/msg01490.html>

- *From:* LCC <claylc@xxxxxxxxxxxx>
 - *Date:* Mon, 23 Jul 2007 01:13:11 -0700
-

Christmas in July! Santa Claus comes early this year!

If you have never heard of me before, then I respectfully suggest that you should read on my Google group "Lonnie Courtney Clay" the thread titled "I launch a total war preemptive nuclear strike against hackers"

I pride myself on being an exceptional con artist as well as extremely intelligent.

I first posted on Dejanews (now Google) in 1997 as tc5526@xxxxxxxx then in 2001 on Shrapnelgames forums as LCC – see postings of Anonymous, then on Google groups in 2001 as lcclay1@xxxxxxxx This year on Apr 17, I started posting on Google groups as claylc@xxxxxxxxxxxx

So far this year I have posted a book worth of text and gained nothing but the increased hostility of others. I hope to turn the situation around with this document. So as well as posting it, I am sending it out by email to various parties in the hopes of "Like a whale from the deep, I rise to to the surface!" gaining allies....

OLD TEXT

Yesterday on 20th July, I went to bed about 4AM feeling well. When I woke up about 6AM, I had an intense pain in my right side, presumably from my liver. As time passed the pain diminished. Also yesterday I visited the VA to discuss the situation with my psychiatric doctor, due to becoming violently ill after taking my prescriptions on July 19th. I decided to fall into non-compliance (according to him) with the court order for mandatory outpatient treatment by refusing to take any medications other than the prescribed anti-psychotics. So I might be ordered institutionalized yet again. I thought that I could recover from whatever was the problem causing the pain around my liver, but such seems to be unlikely now, because today (21st at 9AM) when I started typing it has intensified. I expect that I will be forced to enter the hospital again. I hope to mitigate the antagonism exhibited

Merry Christmas! Linux RULES! New applications to develop!

by the staff on my previous visit by doing something useful before going in. I am typing this up as a quick example of a technique applicable for text messages and files to demonstrate that I can be a useful employee rather than just a con artist. I have chosen to post this document as soon as I finish it. This is only one of a cafeteria plan of techniques which I have conceived over the past twenty plus years to process data into very secure forms prior to applying the battery of standard cryptographic, validation, and error correction techniques to the processed data. It depends upon a shared (between users) set of utilities and dictionary components, and known options to be used for handling the data. Physical security to protect each computer is required, but the compromise of the system by hacking of a computer is mitigated by immediately changing the options used as security breaches occur. If this document breaks any laws, which I doubt, then I will go to prison with the sure knowledge that it will be no worse than living under house arrest at home or in the psychiatric ward.

I hereby declare this specification to be public domain, free of all copyright and patent claims.

Laughing Crazy Coot=anagram TARZAN Chic Logo Guy – first used in June 2007

Lonnie Courtney Clay=anagram Loyal, true innocence – just found out, INSULTING

Components of this selection :

- 1) Library of "words".
- 2) Acronym packer
- 3) Acronym unpacker with disambiguation.
- 4) Archive processor

1) Library of "words"

The library of "words" contains an entry for every "word" likely to be encountered in the data, beginning with the 256 possible values of a byte. I estimate that a million "words" will be about right for a common library. The root "word" for real words is found as in standard dictionaries and every variant possibility has a separate "word" entry. For example run, runs, running and so forth. A "word" may be a composite of two real words separated by a space for very common combinations such as "if the", "for a" etc.

a) Lookup of the dictionary can be performed using a "learned" dynamic memory image of frequently encountered "words", with disk access for rarely used "words". The memory image is initialized from file(s) at the start of each message session. As a user accumulates message traffic, the memory image can optionally be saved to disk after each message session. I leave the details as a TBD. From 1987 until 1989, I developed a tool called Splash – "Selective parenthesis locator and

string handler" which did analysis (of C language function and macro calls found in all files on a disk) to create a memory structure which was very fast for "word" lookup. That memory structure was used to generate a cross reference (of all the function calls found) sorted alphabetically by function name, sub-sorted by the file the reference was found in, sub-sorted by the line of the file on which the reference occurred. Rather than give details of Splash here, I suggest that my dossier should be checked to see if the government already has a copy of the free-ware disks which I mailed out. If not, then I provided all disks to a person named Fred Phillips in 1997, who claimed to be a college student trying to start an Amiga free-ware diskette publishing company. He in turn provided the disks to others. I also provided disks to Fred Fish of Amiga fame at various times, so surely somebody has them. After 10 more years I doubt that the disk copies which I have of the originals are readable....

b) Each entry in the dictionary contains :

..1 The byte string of the "word", whose length is specified by a four byte prefix, and the first byte character to be used for the encoded "word". This provides for the dictionary to include normal and first character capitalized versions of a real word. It also provides for resolution of disambiguation problems by decoupling the first character of the byte string from the first byte of the encoded "word". For encoding lookup purposes a dynamic data structure needs to be generated which gives fast recognition for byte strings whose first byte does not match the first byte character in the encoded word. As an example, no match is found for a source byte sequence in the dictionary. Then a check is made using a table of 256 chain header addresses whose entry is selected by the first byte of the sequence. The attached chain of exception "words" points to dictionary entries which are exceptions, with quick recognition made possible by providing in each entry the second through fifth bytes of the sequence of the "word".

..2 The longitudinal redundancy check of the "word" – a byte value.

..3 The shifted longitudinal redundancy check of the "word" – a byte value. The circular shifting of bits used for a particular dictionary can vary from standard (such as one bit shift left circular) to produce custom dictionaries. This is the first of many options available to enhance security.

..4 A byte containing eight different (CRC-16) parity bits (for the two bytes of ..2,3) calculated from a cafeteria of cyclic redundancy check techniques. Which bit of the byte is used for which technique must be agreed upon for users to share a common dictionary. This byte provides not only disambiguation but also the option to perform a high reliability integrity check on the library. First the two LRC bytes are verified, then they are used to verify the byte string of the "word". There may be hundreds of millions of computers using this

algorithm, so it is desirable to have one common dictionary for low security data. However there should be a different version for secure business (and even departmental) traffic to avoid vulnerability to hacking of the entire user network at once. Furthermore, the choice of which byte (of .2 and .3) comes first in each 16 bit composite can be different for each of the eight bit slots – big endian versus little endian, which gives an enormous number of variations possible beyond the 256 possible values of a byte. According to a Wikipedia article, there are at least 26 CRC algorithms. $26!/18!$ is a quite a big number, making it rather difficult to guess what techniques are being used on the 256 possible endian variations (for the values of .2 and .3). So as I said, security can be quite rigorous if a user wants to construct a custom library.

Notes :

If a dictionary creation utility is provided then anybody could create custom dictionaries used for private files or just a few users. As an example, a variant of a standard dictionary could be generated which changes the SLRC shifting, the CRC technique combination, and the endian combination to produce a new set of codes for bytes 3–4 of each code specification for a standard library "word" set.. The utility could also crunch the archive specific dictionary created by LCC–zip specified below in part 4 as a seed of definitions rather than using the standard library of "words". The first high quality product to market could be a real GOLD MINE!!!

To improve execution time for decoding a dynamic data structure is used which begins with a 64k entry table giving subsidiary chain header addresses. The table is indexed by the first two bytes of the four byte specification. The subsidiary chain attached to a table entry gives the third bytes in use and each third byte entry is accompanied by a subsidiary chain header for fourth byte lookup. The fourth byte chain members point to the data for the relevant library "word".

Discussion :

a) A "word" is uniquely identified in a particular dictionary by the dictionary first byte character for the encoded "word" and the 16 million possible combinations provided by the three disambiguation bytes (.2,.3..4).

b) For alphabetical first characters (real words) a further disambiguation is available by the choice of upper or lower case for the first letter, giving 8 million combinations. The upper case should be reserved for conflicts not sufficiently disambiguated by a) above. If a capitalized word is encountered, then it is first checked to see if the dictionary has a matching entry. If not then the capitalized word is represented by a double four byte specification. The first

Merry Christmas! Linux RULES! New applications to develop!

specification says to capitalize the word indicated by the second specification. This is a tradeoff between coded file bloat and dictionary bloat.

c) Single byte "words" such as non-alphabetical special characters and numerals require four bytes to specify them, bloating the coded data by a factor of four. But the average "word" length for real words (including a space between words) usually exceeds four, resulting in compression of the total data stream. For common multiple byte special character combinations (such as a period followed by newline) the dictionary can include the combination as a defined "word".

d) Standard data features such as logos, headers, footers, and warning labels can be included in the library using a special first byte "word" and three byte disambiguation to result in high compression for pre-defined standards. In this case the byte string entry in the dictionary may contain anything at all such as HTML coded images to give a single example. This is a TBD for dictionary designers. See f) below.

e) During decode a space after a real word is implied when the real word is immediately followed by another real word. Real words are recognized by beginning with an alphanumeric character as the first byte of the text string for the "word". If a "word" in the library already includes a trailing space for some reason, then the automatic space generation at decode is suppressed.

f) A first byte code specification value of zero (null) is used to denote such text formatting functions as capitalization of the next word in the acronym, and other text formatting codes, with 16 million combinations available by the three associated disambiguation bytes, which are NOT generated as in b .2,.3,.4 above but are instead defined by the dictionary designers to meet all anticipated requirements. I have no intention of re-inventing HTML here, so what text formatting codes should be included (in what combinations) I leave as a TBD for designers.

..1 One of the requirements may be the regular presence of numeric strings in source data so often that the bloat must be avoided. In such a case the first byte of the specification code of null denotes special function, the first disambiguation byte denotes a numerical byte string, and the remaining two bytes specify the length in bytes (up to 65535) of the numerical byte string. This can be used to result in compression rather than bloat because such packing techniques as binary coded decimal can be used to replace the ASCII text of the source data.

..2 Rather than provide a data interceptor with nice fat strings of numerical data in unencrypted form, the following option can be used. Have a key disk for the user which decodes BCD coded strings according to a rotating mapping of the 16 possible hexadecimal codes to each of

the ten decimal digits. As each hexadecimal half byte is processed, the mapping changes to the next according to the key disk lookup table which is indexed by a pseudo random number generator whose seed is provided by a message specific encryption key—code seed for the PRNG. I leave the details as a TBD, including how to inform the recipient of the relevant encryption key—code.

..3 For byte strings of mixed alphabetical and numeric characters which do not form any of the standard "word" entries in the library, a dumpster catch all solution is to use a null/zero, followed by a first disambiguation code byte denoting a mixed string, followed by two bytes giving the string length. The disambiguation code byte specifies for the particular library whether the byte string is given in the clear or processed first by some scrambling technique. The scrambling technique can be (shared) encryption key controlled or something as simple as a time stamp defined rotating substitution table for the common library. I leave the details as a TBD.

2 Acronym packer

a) Each processed data stream is prefixed by a 64+ bit time stamp in the clear which uniquely specifies the behavior of the acronym packer for the particular processed data stream for the particular released version of packer and library of "words". For messages between secure users the time stamp should be processed using an agreed upon 64+ bit encryption key to result in a modified time stamp which is used instead. I leave the details of encryption key generation (presumably from an encryption key string) such as whether it should be generated from the key string depending upon the time stamp in the clear (and how the time stamp in the clear is combined with the resulting encryption key) up to "experts". I simply declare "LET THERE BE a 64+ bit time stamp", and behold, it SHALL exist....

b) The acronym packer processes the input stream of "word" four byte specifications using the time stamp to scramble the order of the output data stream to be truly FUBAR. When a specification is encountered for a variant "null" byte stream as in f .1,.2,.3 then that specification cuts short the preceding block of "word" four byte specifications and is handled by itself in the input processed data stream specification sequential position without further scrambling. Blocking then resumes until another "null" byte stream or end of message is encountered. The first byte of each four byte specification ALWAYS comes first. A time stamp and input data stream sequence controlled rotating reordering of the other three bytes for each four byte specification can be used as a first scramble. I leave how the rotation is controlled as a TBD, but it should be non—trivial – see c) for an example. An overview processor for the data can be used to chop it into blocks between null variants in a deterministic manner.

c) For each block (of four byte specifications), the block length in

number of specifications and how the block members are scrambled is a library designer TBD and time stamp controlled to maximize confusion. For example, the time stamp can be used to generate a table of varying block lengths which rotates in a manner similar to the order scrambling tables described below. The time stamp can be used to generate (input data stream specification) order scrambling tables for each block length which are nested to any desired depth depending upon data stream length. As an example, blocks may be 1, 2, 4, 8, 16, 32, 64, 128, 256 (of four byte specifications) in length. The lowest level table would be a simple order scramble giving the index in the original stream (for the implied table index) of the processed stream. Each entry of the table for the second level up would specify a direction of rotation and number of entries to rotate the first level table after each block is processed. The second level table has 256 entries and is completely used once with wraparound from one end to the other when necessary. The third level table and on up the chain for a while operates on the table below it to specify a direction of rotation and the start index for pulling table entries from the table below it. At the top is a table of 256 PRNG seeds which is derived from the current time stamp (being processed by a TBD method) which might be as simple as (a library designer or key-disk specified) direction and circular rotation bits to shift for the current time stamp to generate each entry. Alternatively it might be a bit scramble of the preceding time stamp entry in the table. I am getting bored and am in pain, so as I said above call it TBD. Using the table of seeds, (for each seed as it is indexed) the lowest level table is regenerated by using a scratchpad diminishing linked list of entry numbers not yet pulled and taking the latest random number modulo number of entries remaining. Techniques such as this should make scrambling of multiple gigabyte messages follow no apparent rhyme or reason and should be unbreakable. If not then I will come up with something more elegant when i am not ill...

3 Acronym unpacker with disambiguation

Do I really need to explain this ? Maybe later...

I was going to hold this in reserve, but I have decided to fire off all of my guns at once, so here goes nothing.

4 Archive processor

I am going to re-invent zip, which is a thoroughly mature application. I think that I understand how it works as applied to processing of archives. If not, then I will revise this later. I am giving this specification because I want zipped files to use the library of "words" described in item 1) above to result in coded zip files. I call my version LCC-zip. For a particular run of LCC-zip, there is one pass through all input files to develop the LCC-zip dictionary. That

Merry Christmas! Linux RULES! New applications to develop!

pass is followed by a second pass through all input files to generate packed files giving the LCC–zip dictionary and the compressed data stream utilizing the run specific LCC–zip dictionary. Output file formats I leave as an exercise for the student.

Optionally that pass is followed by usage of the "library of words" described in item 1) above to replace the clear text definitions in the LCC–zip dictionary by library defined four byte specifications so that the source is not only compressed but also securely coded. If a LCC–zip dictionary entry is a composite of multiple library "words" then a sequence of four byte specifications from the library is given rather than just four bytes. For standard features {as specified in 1) discussion item d above}, the body of the zipped results is searched for first byte codes of 0xc0 (specifying not tied to the dictionary) and when found the byte string is compared against the library for a match. Comparison can go very quickly because the lengths of the byte string and the library object must be equal. If a match is found, then the entire 5+n byte sequence is replaced by a five byte sequence. The first byte is 0xff and the next four bytes are provided by the library. This could result in enormous compressions for well constructed dictionaries..

For unzip you simply replace the library codes in the LCC–zip dictionary by the clear text definitions from the library before processing the zipped files. As the zipped files are processed they are searched for first bytes of 0xff. When 0xff is found the library is used to restore the original byte stream. It gives a lot of pain to snoopy people at very low cost.....

a) LCC–zip creates a custom dictionary for the specific set of packed files which uses 2 to 5 bytes to specify a (processed) byte string code for each common source byte sequence found in the archive.

..1 The string code first byte most significant two bits give the number of additional bytes (1–4) required for disambiguation. The most common 16k source byte sequences require only two bytes. Three bytes give the most common 4 million, four bytes for 1 billion, and finally five bytes for 256 billion, which should be rare indeed.

..2 In the case of unique source byte sequences such as are found in the HTML objects for graphics and sound, there is no dictionary entry. Instead of that, the first byte code of 0xc0 is always followed by a four byte count of bytes in the sequence then by the unprocessed source byte sequence. The code 0xc1 could be used with a four byte compressed string byte count to convert long ASCII numeric strings to BCD packed format. The LCC–zip which is developed needs to have the ability to decode at least HTML formatted files in addition to text files. Whether it supports smart processing of other file formats is up to the code developers. I am NOT going to write this thing, since doing so is an exercise for journeymen rather than masters such as I.

b) In the first pass all files are scanned to create a dynamic memory structure similar to my Splash which has a 64k address root table indexed by the first two bytes of the source data sequence in the current window of scan. Each entry in the root table has an associated four byte counter which is pegged at $2^{32}-1$ to avoid overflow. The maximum width of the current window of scan (up to 255 bytes) is decided by the user at run time and determines how much execution time is consumed in the first pass. I recommend 48 bytes maximum width as a default. If the LCC-zip has smart analysis of file components, then just process for dictionary construction windows of scan confined to text data streams.

The current window of scan begins with either of five cases :

..1 If the entire current window of scan processing is filled with ASCII numeric characters or space/blank characters, then increment the start index for the next window of scan by the maximum width of the window of scan and end processing for the current window of scan.

..2 An alphanumeric character following a non alphanumeric character :
In this case the current window of scan processing is truncated to the first character of the last block of non alphanumeric characters in the current window of scan, provided the current window of scan contains any non alphanumeric characters. Notice that this allows for multiple real word combinations to be processed, but does not include real word fragments chopped off by the current window of scan's end. It also usually (for text) terminates the current window of scan with either a space/blank or punctuation. After processing of the current window of scan, the start index for the next window of scan is set to the first non alphanumeric character in the current window of scan. If all of the characters are alphanumeric, then the start index is just incremented by one.

..3 An alphanumeric character which follows an alphanumeric character :
In this case the current window of scan processing is truncated to the first non alphanumeric character found in the current window of scan. If all of the characters in the current window of scan are alphanumeric, then after processing the start index used for the next window of scan is just incremented by one. Otherwise the start index for the next window of scan is set to point to the first non alphanumeric character in the current window of scan. Notice that this provides for processing to move past the tag end of long alphanumeric sequences as soon as a non alphanumeric character is encountered.

..4 A non alphanumeric character other than space/blank :
In this case the current window of scan processing is truncated to the character preceding the first character found in the current window of scan which is alphanumeric. After processing of the current window of scan, the start index used for the next window of scan is just incremented by one. Notice that for the very common case of ". " this results in the next window of scan starting with a space/blank. A

punctuation character following a real word sequence is included in the processing of the real word sequence itself.

..5 A first character of space/blank :

In this case the processing of the current window of scan is truncated to the character preceding the first non space/blank character in the current window of scan. If the result is a single space/blank then the current window of scan processing ends after incrementing the index for the next window of scan by one. In the case of multiple space/blank strings the index for the next window of scan is set to point to the first non space/blank in the current window of scan. If the entire current window of scan is space/blanks then the index used for the next window of scan is just incremented by one. Since the root table is indexed by the first two bytes of the current window of scan, such strings as ". " are already covered by the preceding window of scan processing anyway. Note that for the typical case of two words separated by a space/blank, the space/blank is already processed as a terminating character for the preceding word sequence.

Discussion : It will usually occur that a real word is processed with the current window of scan including the real word followed by a space/blank or punctuation. So there will be a data entry in the dynamic memory structure which gives the real word and whose subsidiary data elements are sub-strings continuing the word with a single character such as " " or "," or ".". In such a case the dictionary will generate an entry which includes the punctuation only if it is commonly occurring.

c) The window of scan processing has two input components, a string length and a byte string sequence which starts at the current window of scan processing index. If the string length is one, then the sequence has already been processed by the preceding window of scan, so no further processing is required. Otherwise the first two bytes of the sequence are used to index the root table and increment its associated counter provided that an increment does not result in overflow. If the string length is two then processing is completed. Now check whether the Otherwise the address in the root table entry is either still null or it points to a dynamically allocated data element whose structure type is one of three possibilities and which contains the following data :

..1 A byte value identifying the type of structure for the data element which is selected from the following :

..1.1 – Zero for type zero "header" data elements initially attached to any of the 64k slots of the root table when the first 3+ length byte string sequence references the header table entry. Each header specifies a sub-string whose byte sequence (including the initial third position byte value) down to the current byte string sequence end point is initially found to be a unique sub-string sequence appended to the preceding two byte sequence which determines the

header's parent slot entry. If an alternative variation for the sub-string sequence is found, then the header is revised to specify a shorter sub-string ending just before the fork point byte and to point to a subsidiary element chain head (first of two new type one data elements) which begins a new linked list giving sub-strings beginning at the fork point byte onwards. If the fork point is at the first character of the header's sub-string (byte three of the total sequence), then the header becomes obsolete and the header's parent entry is revised to point to a new type one subsidiary data element chain head and the subsidiary data element points back to the obsolete header's parent entry. In either case the chain head sub-string (beginning at the fork point byte) is inherited from the header and its counter is initialized from the counter of the header. For the second type one entry in the new linked list the counter is initialized to one and the sub-string is the newly discovered alternative sequence beginning at the fork point byte and continuing to the end of the current byte string sequence. If a header already has a subsidiary element connected and a variant is discovered for its sub-string, then the subsidiary for the header's new subsidiary linked list header is initialized to point to the old header subsidiary element. The new linked list header element has a sub-string inherited from the header which begins at the fork point byte and includes the tag end of the header's old sub-string. The new variant element's subsidiary is not yet established, so its address is initialized to null. The data element of the new variant has a sub-string from the fork point byte to the end of the current byte string sequence. The subsidiary address of the header is revised to point to the new chain head and the chain head's parent link points back to the header. In all cases of headers the byte string sequence first two bytes provide the root table entry containing the address of the header and the referenced header points back to the entry in the root table.

..1.2 – One for a linked list of type one data elements each of which gives a next link address for the list chain. Each member after the head has a parent address pointing back to the previous chain link. The chain head parent address points back to the parent which specifies the prefix for the chain members' sub-string sequences which begin at the fork point byte index. Each member of the linked list specifies a sub-string beginning at the chain fork point byte index and may give the address of a subsidiary data element which specifies how the byte sequence continues after the sub-string of the type one data element. The subsidiary data element pointed to can be either a type one chain head, or a type two table element. When the linked list length exceeds TBD different (index of fork point byte values) type one data elements, then the linked list becomes obsolete and is replaced by a type two data element. The chain's affected type one elements all undergo revision or removal to take into account the new prefixing byte value determined by the entry in the type two data element. If the sub-string of an affected type one data element is of length one, then it becomes obsolete and its subsidiary element address is placed into the table entry of the new type two data

element specified by the single character sub-string value of the now obsolete type one data element. Otherwise its sub-string start pointer is advanced one byte and it becomes the head of a single element new linked list chain attached as a subsidiary to the new type two data element at the entry of its old sub-string first byte character value. In either case when a new type two entry is created, the frequency counter is initialized for the table entry to the value from the matching type one data element. If a variant of the type one element's sub-string is found and the variation occurs after the fork point byte index, then a new subsidiary chain is formed whose head is pointed to by the affected type one element's subsidiary address. The head entry frequency counter is initialized from the parent and the new variant counter is initialized to one. Each type one entry specifies a continuation sub-string which is prefixed by all of the parent entries up the structure to form the byte sequence of the full string by appending the type one entry's sub-string and (if a subsidiary link is active) specifying the variations of the byte sequence which are suffixes to the type one element's sub-string.

..1.3 – Two for a type two data element which gives a table of 256 data element subsidiary addresses and 256 associated four byte counters protected from overflow of course. This is the most memory intensive of the dynamically allocated data elements. On the other hand it speeds up execution time significantly when compared to searching linked lists. Trade offs are a basic fact of life....

..2 A byte value giving the character count n in the data element's byte sub-string sequence.

..3 A byte giving the first character in the byte sub-string sequence for the current data element, except in the case of type two data elements.

..4 a byte value – PAD

..5 A pointer to the parent which is accessing the structure.

..6 One of the following blocks of information :

..6.0 – for type zero data elements,

..6.0.1 A four byte frequency counter which is incremented each time that the current structure is found to match the byte string sequence, protected from overflow of course.

..6.0.2 A pointer to a dynamically allocated memory image sub-string start character.

..6.0.3 A pointer to the address of a subsidiary data element.

..6.1 – for type one data elements,

..6.1.1 A four byte frequency counter which is incremented each time that the current structure is found to match the byte string sequence , protected from overflow of course.

..6.1.2 A pointer to a dynamically allocated memory image sub-string start character.

..6.1.3 A pointer to the address of a subsidiary data element.

..6.1.4 A pointer next link address for connecting list members.

..6.2 – for type two data elements,

..6.2.1 A table of 256 pointers to the address of the subsidiary data elements.

..6.2.2 A table of 256 four byte frequency counters.

d) The window of scan processing begins a walk through the dynamic memory data elements with access of the entry of the root table indicated by the first two bytes of the current byte string sequence.

..1 If the table pointer is still null, then a new type zero header element is created with initial count of one. The current byte string sequence is copied from the current file's memory image onto a dynamic memory stack so that it remains valid after the next file is loaded. The item .6.0.2 start sub-string pointer is set for the initial sub-string of the current byte string sequence beginning at the third character of the sequence. The subsidiary link is set to null and the parent link points back to the selected entry of the root table. The sub-string character count is initialized to be two less than the total length of the current byte string sequence. The first character is copied from the current byte string sequence third character position.

After getting up from a nap and seeing my father for his birthday party, I sat down again to continue this document. I knew that I had taken twice as long for part 4 Archive processor to be typed up so far as for all of parts 1–3 combined. Now I notice that part 4 is half the size of the total document, with no clear end in sight. So I checked what I had typed and concluded that I was suffering from diarrhea of the mouth. So I am going to stop the micro-management explaining and just give some pointers for the presumably master developers who may be reading this document. The dynamic memory allocated data structures are sufficient tutorial for experts to deduce how to write the program.....

A) Dynamic memory allocation is essential for this application. For Splash I created a master scratchpad memory manager which got the contiguous memory chunks whenever I ran out, including an oversight function which was called each time anybody wanted to put something on the scratchpad which was the only caller of the master scratchpad memory manager. The number of bytes requested was checked against the remaining current contiguous block free space, and appropriate action taken such as returning an address in the current block or making the call for a new block before returning an address at the start of the new block.

B) In the development of Splash I tried out garbage collection of the dynamic memory allocated data which was made obsolete. On a two megabyte machine, I determined that it took as much memory for the code to handle GC as what the GC code recovered. However on a gigabyte plus machine GC is worthwhile and should be considered as a fundamental part of the plan.

C) After all of the files are scanned, the entire dynamic memory structure should be stepped through to determine which data elements have counters of $n=3+$ (TBD) or more for a byte count of the byte sequence string (from the root table to the end of the sub-string of the current data element) of four or more. Have a four byte master counter which is incremented by $9+$ (byte sequence byte count) for each data element meeting the test. (In the case of byte counts under four, all of the combinations with counts of $m=5+$ (TBD) are allocated to the first four million codes. So there is little to manage in the way of sophistication for short strings.) If the byte master counter for the passing data elements found would exceed the free memory available during the next step D) below, then increment the required minimum counter value and re-scan the entire dynamic memory structure until memory is sufficient.

D) Dynamically allocate a dictionary table of size 256 which is indexed by sub-string byte count and which is a list of chain head entries for dynamically allocated chains whose members give a next link address and a pointer to a null terminated text string. Also dynamically allocate an associated table indexed by byte count which contains (four byte) counters to be incremented each time a member is added to the head of chain for the entry. Once again step through the entire dynamic memory structure and this time when the byte count requirement is met, add the string with null terminator to a dynamic scratchpad. Add a new member to the head of the chain for the appropriate dictionary table entry (given by the byte count) and set the text string pointer of the new member to the start of the new string in the dynamic scratchpad.

E) All of this work in memory is vulnerable to loss if the power glitches. So it is about time to create a dictionary file. The file is divided into sections of increasing byte count with each section giving a list of null terminated strings. Each section has a header giving byte count followed by the list. Then a footer such as sixteen `0xff` or some other recognizable code is used to terminate the section. Start the file list off by writing to file the two byte sequences implied by the entry number in the root table (which meet the minimum count requirement), following each two byte code by a null terminator. The next section is three byte codes. Step through the root table and process each attached subsidiary data entry to find all three byte codes (which meet the minimum count requirement). As each code is found write it to file with a null terminator.

F) When I say "write it to file" I really mean to a file scratchpad of reasonable size such as 32 megabytes. Managing the file scratchpad I leave as an exercise for the student. So far the codes written result in very little data compression. They are present to avoid the unseemly situation of a short byte sequence which is not found in the dictionary and therefore must be represented by a five byte 0xc0 code followed by the byte sequence, which would result in bloat rather than compression.

G) Step through the dictionary table starting with the entry for a byte count of four. Use the associated chain member counter to determine whether you will have an overflow problem in your file scratchpad and manage accordingly. Beginning at the chain head, step through the chain using the pointer to null terminated text string to extract and write to file each of the text strings from the chain. As each chain of a particular byte count is processed, write the file scratchpad to disk.

H) When the disk file is complete, free up all of the dynamically allocated memory because we are starting over!!! Reallocate the root table of 64k addresses and this time you need a table of 64k two byte codes rather than counters. There is just one type of data element in the dynamically allocated memory structure now. It contains the following :

..1 Parent link address for the current entry in the chain. For chain headers this points to the parent at the next level up rather than giving the chain previous link address.

..2 Next link address for the current level chain.

..3 Subsidiary link chain header address for the next level down.

..4 Pointer to a dynamically allocated null terminated text string which gives the entire source byte string sequence for the current element (beginning with the two byte sequence of the root table entry) and which terminates at the character preceding the fork point byte in the source byte string sequence implied by the subsidiary link chain members.

..5 Pointer to the start of the sub-string for the current entry which is prefixed by the parent element sequence (beginning at the root table) and which terminates at the character preceding the fork point byte in the source byte sequence implied by the subsidiary link chain members. This pointer reuses the text string of .4 above and is present as a speedy way to get the start byte of the sub-string so that as the current window of processing steps through a chain (for processing of a particular source byte string sequence) there is a quick way to check whether the current data element sub-string matches the current character sub-string of the source byte string sequence.

..6 A pointer to the dictionary code for the current entry's text string (giving the entire source byte sequence) which is set the first time that the entry is actually used to replace source file text by a dictionary code. Although a parsimonious developer would use all of the two byte codes before proceeding to three byte codes etc., I recommend that instead of doing so the two byte codes should be reserved for two and three byte sequences. The four million three byte codes should be reserved for source byte sequences of byte count five or less. The billion four byte codes should be reserved for sequences of byte count 32 or less. Anything with a greater than 32 byte count should be a five byte code.

I) Read in the dictionary file, creating the dynamically allocated memory structure entries by stepping through the structure beginning at the root table.

..1 If the dictionary text byte count is two, then pull the next two byte dictionary code and set the two byte code of the indexed entry in the root table. Add the code and its associated null terminated byte string to the memory dictionary. Go back to step I) until the last two byte dictionary entry has been read in.

..2 If the root table subsidiary link is null, then hook up a new element chain header address to the root table and set the new element parent to the address of the root table entry. Go back to step I)

..3 If the dictionary code is a three byte sequence then replace the chain header for the current entry in the root table by the new element. So point the parent of the old header to the new element, the next link address of the new element to the old header, the parent of the new element to the root table entry, and the root table subsidiary link to the new element. Go back to step I)

J) When the root table entry gives a subsidiary link and the byte count of the next file dictionary string is greater than three then beginning with an index of 3 and pointing to the chain head entry given by the root table entry for the first two bytes of the file dictionary string :

..1 Step through the relevant chain of elements whose sub-string begins at byte "index". Because the dictionary was formed to have a counter value for parent elements at least as great as that of subsidiary elements, there should always be an element in some parent chain which has already been read in from the dictionary before a dictionary entry with a text string byte count greater than the parent's.

..2 When the first character of the sub-string for the current element matches the corresponding byte in the text string of the dictionary entry, then check to see if a subsidiary link has already been allocated.

..a If so then check whether the current element sub-string is a match to the byte sequence of the dictionary entry being added. If so then increment "index" and compare it to the byte count of the current file dictionary string. Check whether they are equal then :

..a1 If so then replace the subsidiary link chain header by a new element for the dictionary entry being added. Point the parent of the

Merry Christmas! Linux RULES! New applications to develop!

old subsidiary chain header to the new element, point the next link address of the new element to the old subsidiary chain header, point the parent of the new element to the current level element, and point the subsidiary link of the current element to the new element. Then pull a new file dictionary entry and go back to step J) above.

..a2 If not then point to the subsidiary element of the current element and go back to step .1

..b If a subsidiary link has NOT already been allocated, then check whether the current element sub-string is a match to the byte sequence of the dictionary entry being added.

..b1 If so then create a new subsidiary chain header element. Point the subsidiary link of the current element to the new element, point the new element parent link back to the current element, and set the next link address of the new element to NULL. Then pull a new file dictionary entry and go back to step J) above.

..b2 If not then something has gone wrong ROTFLMAO.

..3 When the first character of the sub-string for the current element does NOT match the corresponding byte in the text string of the dictionary entry, then step forward using the next link address to the next element in the chain.

..a If there WAS no next link then something has gone wrong ROTFLMAO

..b Otherwise go back to step .1

I notice that I have become slap happy with fatigue and I am no longer being very helpful. So it is time for the fat lady to sing and shut this circus down. Just a few more comments and I will post it.

K) When the dictionary has been reconstructed in memory from file, begin Pass 2 scanning of all archive files again. Using knowledge of file formats, process only text data with the dictionary. Use whatever compression technique is appropriate for non text data such as HTML objects for images and audio. For long numeric strings convert them to BCD strings. Using the library of section 1 discussion item d) recognize standard features and replace them by an identifying code which prefixes the four byte specifications from the library. For the text data use a smart algorithm to chop a text stream into blocks of dictionary defined strings which are either contiguous or separated by special character strings to be specially coded as 0xc0 – 4 byte count – string. When each dictionary entry is first used pull an appropriate unused code, set the code for the dictionary data element and otherwise track what is actually used to compress files. When all files have been processed save to file the dictionary which was actually used with the text strings coded by the library of section 1 above.

L) When the archive is to be unpacked, use the library to decode the dictionary and restore coded standard features and otherwise use expert techniques to minimize resources consumed in the unpacking process.

Merry Christmas! Linux RULES! New applications to develop!

Well that was a whirlwind cleaning up the remaining discussion like a white tornado!

Completed at 1:40 AM on July 23, 2007

It took less than two days to produce this document. Frankly speaking I have no good estimate of how long it took to develop the concepts embodied here. However just to give a notion, I spent over 1000 hours developing the free-ware Splash, which is the basis for the part 4 specification. I really hope that others appreciate the value of this document. But as the amateur entertainers say – DONT CLAP – THROW MONEY!!! Send checks to 3395 Harrell Rd. Arlington TN, 38002-4261.

Laughing Crazy Coot/
Lonnie Courtney Clay