Project: Base64 Content-Transfer-Encoding

Due: Tuesday March 12, 2002

Objective

The objectives of this programming assignment are 1) to gain more familiarity with data manipulation at the bit level, 2) to develop further experience using Linux system calls.

Background

Exchanging binary files by email is not quite straightforward because many mail servers were designed to handle text, not binary data. Attempts to send binary files through these servers can result in mangled files. For example, some mail servers might ignore the most significant bit of each byte, since standard ASCII encoding uses only 7 bits. Other mail servers truncate all data beyond the 80th character of each line. In fact, the whole concept of a line is meaningless when we work with binary files. To complicate matters, email is often routed through several servers, so the problem might not be with either the sender's mail server or the receiver's mail server.

The MIME (Multipurpose Internet Mail Exchange) standard defined in Internet RFC 1521 is a comprehensive mechanism for formatting Internet messages.† For many people, MIME is synonymous to email attachments. We are interested in just one section of this standard the Base64 Content-Transfer-Encoding that specifies how binary files should be converted into a text file that can be sent intact through most mail servers. The complete specifications of the Base64 standard are (what else) attached at the end of this project description.

Assignment

Your assignment is to write an assembly language program that prompts the user for the file names of an input file and an output file. The program must transform the data in the input file into a text file in a manner that complies with the Base64 Content-Transfer-Encoding. The output of the program must be stored in the output file.

As a reference standard, we will use the `mimencode` command on `linux.gl.umbc.edu`. Using `mimencode` with the `-u` option, we can convert the output of your program back to binary. If your program works correctly, the output of `mimencode -u` should be identical to the original input file.

For 15% extra credit, write an assembly language program that reverses the process of your first program. I.e., the second program prompts the user for an input file and an output file. If the input file is a properly formatted text file that conforms to the Base64 standard, your program should store the corresponding binary file in the output file.

Implementation Issues:

1. All of the file conversion must be done by your program. You are, of course, not allowed to make a system call to `mimencode`.
2. Files can be opened for reading using a system call to the `open()` function. The C function prototype of `open()` is:

```
int open(const char *pathname, int flags);
```

† Unlike other organizations (e.g., ANSI, ISO) which publish standards with lofty-sounding titles, the Internet Engineering Task Force's (IETF's) standards are for historical reasons published as Request for Comments (RFCs). Although not all RFCs are standards, the specifications of just about every Internet protocol can be found in an RFC. For more information on RFCs and how they are published, check out <http://www.rfc-editor.org>.

According to the Linux system call convention, the syscall number for `open()` should be stored in EAX, a pointer to a null-terminated string with the name of the file to be opened should be stored in EBX and the flag `O_RDONLY` should be stored in ECX. The return value, stored in EAX, is a file descriptor (a 4-byte integer) that can be used in subsequent syscalls to `read()`. Further information on `open()` can be obtained from the Linux man pages. Type `'man 2 open'`.

3. Symbolic constants for syscall numbers, flags, etc can be found in a file called `stddefs.mac` in the directory: `afs/umbc.edu/users/c/h/chang/pub/cs313`. Copy this file into your own directory. Then, the file can be included in your assembly language program using the NASM directive:

```
%include "stddefs.mac"
```

4. To open a file for writing, a syscall to `creat()` is more appropriate. The C function prototype for `creat()` is:

```
int creat(const char *pathname, mode_t mode);
```

   Calling `creat()` is very similar to calling `open()`. The difference is that the file is opened for writing and the file is created if it does not already exist. If a file with the same name already exists, it is overwritten. As before, the return value stored in EAX is a file descriptor. The second argument to `creat()` is used to set the permissions of the newly created file (as in the `chmod` Unix command). You will most likely want to allow the user to read and write to the file, so store the expression `S_IREAD|S_IWRITE` in the ECX register. `S_IREAD` and `S_IWRITE` are defined in `stddefs.mac`.

5. Remember to close all open files before your program terminates. This is accomplished with a syscall to `close()` with the file descriptor as the sole argument. The `close()` function has the following function prototype:

```
int close(int fd);
```

6. Once a file is opened, you can read from and write to it using the `read()` and `write()` syscalls as you have done with `stdin` and `stdout`.
7. Despite what the man pages say, you can tell that you have reached the end of a file you are reading when `read()` returns 0.
8. Recall that `read()` stores the characters read at the address provided and returns the number of characters read. The string read in is not null-terminated. Also, if the string is read from `stdin`, the last character is a `'\n'`. Thus, some massaging of the string is needed before it can be used as a file name.
9. You should not assume that the file has run out of bytes when `read()` does not return the maximum number of bytes requested.
10. It is inefficient to read 3 bytes at a time.
11. The functions `open()`, `creat()` and `read()` return the value -1 if an error is encountered. The cause of the error is given as an error code in the global variable `errno`. If you wish to examine these values, you must declare `errno` to be an external label. Symbolic names for some of the possible values for `errno` can be found in `stddefs.mac`. Consult the Linux man pages for the meaning of each error. If you reference `errno`, then you must link your program using `'gcc -nostartfiles'` instead of `ld`.

12. Recall that the Intel Pentium CPU is little endian. If you move multiple bytes into a register, the bytes might not be ordered the way you like.
13. Assembly language instructions that you might find useful include: AND, OR, SHL, SHR, XCHG.
14. A common task that you will want to perform is: add a new character to the output buffer, then write out the buffer if it is full. You will probably want to write a subroutine to do this. Invent your own parameter passing conventions.
15. Read the Base64 specifications for handling the last few bytes of input carefully. The output may need to be padded with 1 or 2 `'='` as appropriate.
16. If you want to have your output appear identical to the output from `mimencode`, print out 72 characters per line.

Turning in your program

Use the UNIX `'submit'` command on the GL system to turn in your project. The class name for submit is `'cs313'` and the project name is `'project'`. Sample runs and a typescript file is not needed for this project. The grader will simply test your program using `mimencode` and some binary files. Include a README file if your submission needs any special attention.

References

1. Borenstein, N. and Freed, N. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies." RFC 1521, September 1993. Available at <ftp://ftp.isi.edu/in-notes/>.

5.2 Base64 Content-Transfer-Encoding

The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable. The encoding and decoding algorithms are simple, but the encoded data are consistently only about 33 percent larger than the unencoded data. This encoding is virtually identical to the one used in Privacy Enhanced Mail (PEM) applications, as defined in RFC 1421. The base64 encoding is adapted from RFC 1421, with one change: base64 eliminates the "*" mechanism for embedded clear text.

A 65-character subset of US-ASCII is used, enabling 6 bits to be represented per printable character. (The extra 65th character, "=", is used to signify a special processing function.)

> *NOTE: This subset has the important property that it is represented identically in all versions of ISO 646, including US ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. Other popular encodings, such as the encoding used by the uuencode utility and the base85 encoding specified as part of Level 2 PostScript, do not share these properties, and thus do not fulfill the portability requirements a binary transport encoding for mail must meet.*

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single digit in the base64 alphabet. When encoding a bit stream via the base64 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first byte, and the eighth bit will be the low-order bit in the first byte, and so on.

Each 6-bit group is used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string. These characters, identified in Table 1, below, are selected so as to be universally representable, and the set excludes characters with particular significance to SMTP (e.g., ".", CR, LF) and to the encapsulation boundaries defined in this document (e.g., "-").

Table 1: The Base64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The output stream (encoded bytes) must be represented in lines of no more than 76 characters each. All line breaks or other characters not found in Table 1 must be ignored by decoding software. In base64 data, characters other than those in Table 1, line breaks, and other white space probably indicate a transmission error, about which a warning message or even a message rejection might be appropriate under some circumstances.

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 24 input bits are available in an input group, zero bits are added (on the right) to form an integral number of 6-bit groups. Padding at the end of the data is performed using the '=' character. Since all base64 input is an integral number of octets, only the following cases can arise: (1) the final quantum of encoding input is an integral multiple of 24 bits; here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding, (2) the final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by two "=" padding characters, or (3) the final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be three characters followed by one "=" padding character.

Because it is used only for padding at the end of the data, the occurrence of any ´=' characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three.

Any characters outside of the base64 alphabet are to be ignored in base64-encoded data. The same applies to any illegal sequence of characters in the base64 encoding, such as "====="

Care must be taken to use the proper octets for line breaks if base64 encoding is applied directly to text material that has not been converted to canonical form. In particular, text line breaks must be converted into CRLF sequences prior to base64 encoding. The important thing to note is that this may be done directly by the encoder rather than in a prior canonicalization step in some implementations.

> *NOTE: There is no need to worry about quoting apparent encapsulation boundaries within base64-encoded parts of multipart entities because no hyphen characters are used in the base64 encoding.*