Echo stdin to stdout

This is your first assembly language project. The program, befitting a first project, is straight forward enough. In a loop, it reads 1 byte from **stdin** and writes it to **stdout**. See below for how the program is made to exit.

Using man pages

Below you will find references using and reading *man* pages. This is how the early Unix system supplied their documentation and these are still the definitive authority.

Warning: don't use man from the Macintosh terminal!

Apple changed some function signatures and meanings. Use man only from the WSL or Linux command line to be certain you're getting the right information.

Invocation

From the keyboard

If you simply run the program, it will read your keystrokes. Nothing will be output to you. When you hit enter, the line you typed will be written back.

Execution stops when you:

a) hit ˆc (control C) - this sends a signal to the receiving process which usually kills it.

b) you enter an end-of-file - ˆd (control D) on Linux.

Sample:

```
a@PROMETHEUS:~/repos/pk_echo$ ./a.out
Foo
Foo
a@PROMETHEUS:~/repos/pk_echo$
```

I invoked the program and typed "Foo" and hit enter. The program wrote back "Foo" and a new line. Then I hit ˆd so the program exited.

From redirection

An awesome feature of the command line is the ability to redirect input and output. Since the program simply reads from **stdin**, and **stdin** can be redirected, you get this feature for free:

```
a@PROMETHEUS:~/repos/pk_echo$ ./a.out < README.md
# pk_echo
A COMPORG project suitable for 1st project
a@PROMETHEUS:~/repos/pk_echo$
```

The input to the program came from the file named on the command line after the "<". The less-than says "take input from". My program knows nothing about where its input is coming from. Notice, I did not change it in any way.

`stdin` and `stdout`

You know what `cin` and `cout` are. They are built on top of `stdin` and `stdout`. These in turn devolve downwards into input and output channels denoted by *file descriptors*.

File Descriptor	Name	C++ Equivalence
0	stdin	cin
1	stdout	cout
2	stderr	cerr

File descriptors are small integers which are in fact, array indices into an array of open "files" maintained by the Operating System. They can range from 0 upward to a small limit such as 20 on old Unix systems to 2560 on the current Mac OS and 1024 on the current Linux. The size of the array defines the maximum number of open "files" a process can have at one time.

The Apple number is just stupid.

`read()` and `write()`

The lowest level I/O on a Linux system are `read()` and `write()`. Here are their signatures:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

First, let's examine the types and their correspondence to register types.

Type	Equivalence	Register Type
ssize_t	long	x
int	int	w
void *	void *	x
size_t	unsigned long	x

You are expected to use the correct register types.

To get detailed information on these two calls, use the `man` pages.

`man 2 read`

`man 2 write`

From this documentation you can determine the meaning of what they return. The return value of `read()`, if not 1, should cause your program to gracefully exit. A negative return value from `write()` should cause an error message to be printed using `perror()` followed by a graceful exit.

perror()

There is an externally defined integer variable called `errno` that receives error codes from various functions including those in the C runtime library (CRTL). Reading the documentation indicated above will mention `errno`.

The CRTL provides the function `perror()` to print errors. `perror()` interprets `errno` to generate an error message for you. This is the signature of `perror()`:

```
 void perror(const char *s);
```

The argument `s` points to a null terminated C string that is prepended to the error message.

The man page can be accessed here:

```
man 3 perror
```

read() needs a buffer

The second argument to both `read()` and `write()` is a `void *`. This is a generic pointer - it points to a location in memory that's considered just bytes.

In your `data` segment, I suggest declaring a fixed buffer. There's no reason not to make it 8 bytes long knowing that you will only ever fill one byte.

Special registers

Remember that `x29` and `x30` are special.

Return value from the *program*

Programs can act like functions to other programs. This is why you have always returned 0 from `main()` (or were taught about other values if you were taught by me).

If no errors are detected on `write()`, you must return 0.

If an error is detected on `write()` you must return 1.

Any error detected on `read()` should end the program and it is OK to return 0.

Other reminders

- Remember to have a `main` marked as global.

- Remember to `.align`. A more "current" version of this is to use `.p2align`. Use that instead. The argument is the exponent to a power of 2 to which to align. Thus, `.p2align 2` is typical.

- Remember to have a `.text` segment

- Remember to end your program with `.end`

Work rules

All work is done alone. No partners.

Expectations

The following is provided to set your expectations and is not a challenge. Including some comments and blank lines, my Apple / Linux solution runs 40 lines. You have one week to write about 40 lines plus one grace day. You should be able to crush this if you ask questions and read my book.

Helpful video

There is a video on Youtube demonstrating `read()`. The source code is written for Linux only, but this shouldn't be a problem. Find the video on Schoology.

The first person to ask "Where on Schoology" earns my scorn.