Porting Newlib

From OSDev Wiki

Newlib is a C library intended for use on embedded systems available under a free software license. It is known for being simple to port to new operating systems. Allegedly, it's coding practices are sometimes questionable. This tutorial follows OS Specific Toolchain and completes it using newlib rather than using another C Library such as your own.

Difficulty level

Advanced

Porting newlib is one of the easiest ways to get a simple C library into your operating system without an excessive amount of effort. As an added bonus, once complete you can port the toolchain (GCC/binutils) to your OS - and who wouldn't want to do that?

Contents

Introduction

I decided that after an incredibly difficult week of trying to get newlib ported to my own OS that I would write a tutorial that outlines the requirements for porting newlib and how to actually do it. I'm assuming you can already load binaries from somewhere and that these binaries are compiled C code. I also assume you have a syscall interface setup already. Why wait? Let's get cracking!

Preparation

Download newlib source (I'm using 2.2.0-1) from this ftp server (ftp://sources.redhat.com/pub/newlib/index.html) .

Update: fixed tutorial to work with newer version of newlib.

System Calls

First of all you need to support a set of 17 system calls that act as 'glue' between newlib and your OS. These calls are the typical "_exit", "open", "read/write", "execve" (et al). See the Red Hat newlib C library (http://sourceware.org/newlib/libc.html#Syscalls) documentation.

My kernel exposes all the system calls on interrupt 0x80 (128d) so I just had to put a bit of inline assembly into each stub to do what I needed it to do. It's up to you how to implement them in relation to your kernel.

Porting Newlib

config.sub

Same as for binutils in OS Specific Toolchain.

configure.host

Tell newlib which system-specific directory to use for our particular target. In the section starting 'Get the source directories to use for the host ... case "${host}" in', add a section:

```
  i[3-7]86-*-myos*)
      sys_dir=myos
      ;;
```

newlib/libc/sys/configure.in

Tell the newlib build system that it also needs to configure our myos-specific host directory. In the `case ${sys_dir} in` list, simply add

```
    myos) AC_CONFIG_SUBDIRS(myos) ;;
```

Note: After this, you need to run `autoconf (precisely version 2.64)` in the libc/sys directory.

newlib/libc/sys/myos

This is a directory that we need to create where we put our OS-specific extensions to newlib. We need to create a minimum of 4 files. You can easily add more files to this directory to define your own os-specific library functions, if you want them to be included in libc.a (and so linked in to every application by default).

newlib/libc/sys/myos/crt0.c

This file creates crt0.o, which is included in every application. It should define the symbol _start, and then call the main() function, possibly after setting up process-space segment selectors and pushing argc and argv onto the stack. A simple implementation is:

```
#include <fcntl.h>

extern void exit(int code);
extern int main ();

void _start() {
    int ex = main();
    exit(ex);
}
```

Note: add in argc and argv support based on how you handle them in your OS

newlib/libc/sys/myos/syscalls.c

This file should contain the implementations for each glue function newlib requires.

```
/* note these headers are all provided by newlib - you don't nee
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/times.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <stdio.h>

void _exit();
int close(int file);
char **environ; /* pointer to array of char * strings that defin
int execve(char *name, char **argv, char **env);
int fork();
int fstat(int file, struct stat *st);
int getpid();
int isatty(int file);
int kill(int pid, int sig);
int link(char *old, char *new);
int lseek(int file, int ptr, int dir);
int open(const char *name, int flags, ...);
int read(int file, char *ptr, int len);
caddr_t sbrk(int incr);
int stat(const char *file, struct stat *st);
clock_t times(struct tms *buf);
int unlink(char *name);
int wait(int *status);
int write(int file, char *ptr, int len);
int gettimeofday(struct timeval *p, struct timezone *z);
```

Note: You may split this up into multiple files, just don't forget to link against all of them in Makefile.am.

newlib/libc/sys/myos/configure.in

Configure script for our system directory.

```
AC_PREREQ(2.59)
AC_INIT([newlib], [NEWLIB_VERSION])
AC_CONFIG_SRCDIR([crt0.c])
AC_CONFIG_AUX_DIR(../../../..)
NEWLIB_CONFIGURE(../../..)
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

newlib/libc/sys/myos/Makefile.am

A Makefile template for this directory:

```
AUTOMAKE_OPTIONS = cygnus
INCLUDES = $(NEWLIB_CFLAGS) $(CROSS_CFLAGS) $(TARGET_CFLAGS)
AM_CCASFLAGS = $(INCLUDES)

noinst_LIBRARIES = lib.a

if MAY_SUPPLY_SYSCALLS
extra_objs = syscalls.o # add more object files here if you spli
else                    # syscalls.c into multiple files in the
extra_objs =
endif

lib_a_SOURCES =
lib_a_LIBADD = $(extra_objs)
EXTRA_lib_a_SOURCES = syscalls.c crt0.c # add more source files
lib_a_DEPENDENCIES = $(extra_objs)      # syscalls.c into multip
lib_a_CCASFLAGS = $(AM_CCASFLAGS)
lib_a_CFLAGS = $(AM_CFLAGS)

if MAY_SUPPLY_SYSCALLS
all: crt0.o
endif

ACLOCAL_AMFLAGS = -I ../../..
CONFIG_STATUS_DEPENDENCIES = $(newlib_basedir)/configure.host
```

Note: After this, you need to run `autoconf` in the newlib/libc/sys/ directory, and `autoreconf` in the newlib/libc/sys/myos directory.

Note: `autoconf` and `autoreconf` will only run with automake version 1.14 (exact) and autoconf version 2.64 (exactly) (applies to newlib v2.2.0-1 source pulled on March 10, 2015)

Signal handling

Newlib has two different mechanisms for dealing with UNIX signals (see the man pages for signal()/raise()). In the first, it provides its own emulation, where it maintains a table of signal handlers in a per-process manner. If you use this method, then you will only be able to respond to signals sent from within the current process. In order to support it, all you need to do is make sure your crt0 calls '_init_signal' before it calls main, which sets up the signal handler table.

Alternatively, you can provide your own implementation. To do this, you need to define your own version of signal() in syscalls.c. A typical implementation would register the handler somewhere in kernel space, so that issuing a signal from another process causes the corresponding function to be called in the receiving process (this will also require some nifty stack-playing in the receiving process, as you are basically interrupting the program flow in the middle). You then need to provide a kill() function in syscalls.c which actually sends signals to another process. Newlib will still define a raise() function for you, but it is just a stub which calls kill() with the current process id. To switch newlib to this mode, you need to #define the SIGNAL_PROVIDED macro when compiling. A simple way to do this is to add the line:

```
newlib_cflags="${newlib_cflags} -DSIGNAL_PROVIDED"
```

to your host's entry in `configure.host`. It would probably also make sense to provide sigaction(), and provide signal() as a wrapper for it. Note that the Open Group's (http://pubs.opengroup.org/onlinepubs/9699919799/functions/sigaction.html) definition of sigaction states that 1) sigaction supersedes signal, and 2) an application designed shouldn't use both to manipulate the same signal.

Compiling

You can build newlib in this manner: Newlib is very pesky about the compiler, and you probably haven't built your own i686-myos-gcc toolchain yet, meaning that configure will not be happy when you set target to i686-myos. So use this hack to get it to work (it worked fine for me).

Note: there must be a better way then this.

```
# newlib setup
CURRDIR=$(pwd)

# make symlinkds (a bad hack) to make newlib work
cd ~/cross/bin/ # this is where the bootstrapped generic cross c
                # change this based on your development environm
ln i686-elf-ar i686-myos-ar
ln i686-elf-as i686-myos-as
ln i686-elf-gcc i686-myos-gcc
ln i686-elf-gcc i686-myos-cc
```

```
ln i686-elf-ranlib i686-myos-ranlib

# return
cd $CURRDIR
```

Then run the following commands to build newlib

```
mkdir build-newlib
cd build-newlib
../newlib-x.y.z/configure --prefix=/usr --target=i686-myos
make all
make DESTDIR=${SYSROOT} install
```

Note: SYSROOT is where all your OS-specific toolchains will be installed in. It will look like a miniature version of the Linux filesystem, but have your OS-specific toolchains in; I am using ~/myos as my SYSROOT directory.

For some reason, the newer versions of newlib (at least for me) didn't put the libraries in a location where other utilities like binutils could find. So here's another hack to fix this:

```
cp -ar $SYSROOT/usr/i386-myos/* $SYSROOT/usr/
```

After building all of this, your freshly built libc will be installed in your SYSROOT directory! Now you can progress to building your own OS Specific Toolchain.

Important Note: I found that for newlib to properly work, you have to link against libc, libg, libm, and libnosys - hence when porting gcc, in

```
#define LIB_SPEC ...
```

in gcc/configure/myos.h,

make sure you put

```
#define LIB_SPEC "-lc -lg -lm -lnosys"
```

at the bare minimum.

I highly recommend rebuilding the library with your OS Specific Toolchain after you are done porting one. (don't forget to remove the symlinks, too.)

Conclusion

Well, you've done it. You've ported newlib to your OS! With this you can start creating user mode programs with ease! You may now also add in new functions to newlib, such as dlopen(), dlclose(), dlsym(), and dlerror() for dynamic linking support. Your operating system has a bright road ahead! You can now port the toolchain and run binutils and GCC on your own OS. Almost self-hosting, how do you feel?

Good luck!

Last Updated by **0fb1d8** for compatibility with newer versions of newlib and the OS Specific Toolchain tutorial.

Note: I used a lot of hacks in this article, if you find a better way to do something, please contribute to the page. Thank you.

See Also

Articles

- GCC Cross-Compiler
- OS Specific Toolchain