Genode Porting Guide: Porting an application to Genode's Noux runtime

Porting an application to Genode's Noux runtime is basically the same as porting a program to natively run on Genode. The source code has to be prepared and, if needed, patched to run in Noux. However in contrast to this, there are Noux build rules (*ports/mk/noux.mk*) that enable us to use the original build-tool if it is based upon Autotools. Building the application is done within a cross-compile environment. In this environment all needed variables like `CC`, `LD`, `CFLAGS` and so on are set to their proper values. In addition to these precautions, using *noux.mk* simplifies certain things. The system-call handling/functions is/are implemented in the libc plugin *libc_noux* (the source code is found in *ports/src/lib/libc_noux*). All applications running in Noux have to be linked against this library which is done implicitly by using the build rules of Noux.

As an example on how to port an application to Genode's Noux runtime, we will describe the porting of GNU's `tar` tool in more detail. A ported application is normally referred to as a Noux package.

Checking requirements/dependencies

As usual, we first build GNU tar on Linux/x86 and capture the build process:

```
$ wget http://ftp.gnu.org/gnu/tar/tar-1.27.tar.xz
$ tar xJf tar-1.27.tar.xz
$ cd tar-1.27
$ ./configure
$ make > build.log 2>&1
```

Creating the port file

We start by creating the port Makefile _ports/ports/tar.mk_:

```
LICENSE   := GPLv3
VERSION   := 1.27
DOWNLOADS := tar.archive

URL(tar)  := http://ftp.gnu.org/gnu/tar/tar-$(VERSION).tar.xz
SHA(tar)  := 790cf784589a9fcc1ced33517e71051e3642642f
SIG(tar)  := ${URL(tar)}.sig
KEY(tar)  := GNU
DIR(tar)  := src/noux-pkg/tar
```

As of version 14.05, Genode does not check the signature specified via the SIG and KEY declaration but relies the SHA checksum only. However, as signature checks are planned in the future, we use to include the respective declarations if signature files are available.

While porting GNU tar we will use a dummy hash as well.

Creating the build rule

Build rules for Noux packages are located in *<genode-dir>/ports/src/noux-pkgs*.

The *tar/target.mk* corresponding to GNU tar looks like this:

```
CONFIGURE_ARGS = --bindir=/bin \
                 --libexecdir=/libexec

include $(REP_DIR)/mk/noux.mk
```

The variable `CONFIGURE_ARGS` contains the options that are passed on to Autoconf's configure script. The Noux specific build rules in *noux.mk* always have to be included last.

The build rules for GNU tar are quite short and therefore at the end of this chapter we take a look at a much more extensive example.

Creating a run script

Creating a run script to test Noux packages is the same as it is with running natively ported applications. Therefore we will only focus on the Noux-specific parts of the run script and omit the rest.

First, we add the desired Noux packages to 'build_components':

```
set noux_pkgs "bash coreutils tar"

foreach pkg $noux_pkgs {
  lappend_if [expr ![file exists bin/$pkg]] build_components noux-pkg/$pkg }

build $build_components
```

Since each Noux package is, like every other Genode binary, installed to the *<build-dir>/bin* directory, we create a tar archive of each package from each directory:

```
foreach pkg $noux_pkgs {
  exec tar cfv bin/$pkg.tar -h -C bin/$pkg . }
```

Using noux.mk makes sure that each package is always installed to <build-dir>/bin/<package-name>.

Later on, we will use these tar archives to assemble the file system hierarchy within Noux.

Most applications ported to Noux want to read and write files. On that matter, it is reasonable to provide a file-system service and the easiest way to do this is to use the ram_fs server. This server provides a RAM-backed file system, which is perfect for testing Noux applications. With the help of the session label we can route multiple directories to the file system in Noux:

```
append config {
<config>
[…]
  <start name="ram_fs">
    <resource name="RAM" quantum="32M"/>
    <provides><service name="File_system"/></provides>
    <config>
      <content>
        <dir name="tmp"> </dir>
        <dir name="home"> </dir>
      </content>
      <policy label="noux -> root" root="/" />
      <policy label="noux -> home" root="/home" writeable="yes" />
      <policy label="noux -> tmp"  root="/tmp"  writeable="yes" />
    </config>
  </start>
[…]
```

The file system Noux presents to the running applications is constructed out of several stacked file systems. These file systems have to be registered in the `fstab` node in the configuration node of Noux:

```
  <start name="noux">
    <resource name="RAM" quantum="256M" />
    <config>
      <fstab>}
```

Each Noux package is added

```
foreach pkg $noux_pkgs {
  append config {
        <tar name=\"$pkg.tar\" />" }}
```

and the routes to the ram_fs file system are configured:

```
append config {
        <dir name="home"> <fs label="home" /> </dir>
        <dir name="ram"> <fs label="root" /> </dir>
        <dir name="tmp"> <fs label="tmp" /> </dir>
      </fstab>
      <start name="/bin/bash">
        <env name="TERM" value="linux" />
      </start>
    </config>
  </start>}
```

In this example we save the run script as *ports/run/noux_tar.run*.

Compiling the Noux package

Now we can trigger the compilation of tar by executing

```
$ make VERBOSE= noux-pkg/tar
```

At least on the first compilation attempt, it is wise to unset `VERBOSE` *because it enables us to see the whole output of the* `configure` *process.*

By now, Genode provides almost all libc header files that are used by typical POSIX programs. In most cases, it is rather a matter of enabling the right definitions and compilation flags. It might be worth to take a look at FreeBSD's ports tree because Genode's libc is based upon the one of FreeBSD 8.2.0 and if certain changes to the contributed code are needed, they are normally already done in the ports tree.

The script *noux_env.sh* that is used to create the cross-compile environment as well as the famous *config.log* are found in *<build-dir>/noux-pkg/<package-name>*.

Running the Noux package

We use the previously written run script to start the scenario, in which we can execute and test the Noux package by issuing:

```
$ make run/noux_tar
```

After the system has booted and Noux is running, we first create some test files from within the running bash process:

```
bash-4.1$ mkdir /tmp/foo
bash-4.1$ echo 'foobar' > /tmp/foo/bar
```

Following this we try to create a ".tar" archive of the directory */tmp/foo*

```
bash-4.1$ cd /tmp
bash-4.1$ tar cvf foo.tar foo/
tar: /tmp/foo: Cannot stat: Function not implemented
tar: Exiting with failure status due to previous errors
bash-4.1$
```

Well, this does not look too good but at least we have a useful error message that leads (hopefully) us into the right direction.

Debugging an application that uses the Noux runtime

Since the Noux service is basically the kernel part of our POSIX runtime environment, we can ask Noux to show us the system calls executed by tar. We change its configuration in the run script to trace all system calls:

```
[…]
<start name="noux">
  <config trace_syscalls="yes">
[…]
```

We start the runscript again, create the test files and try to create a ".tar" archive. It still fails but now we have a trace of all system calls and know at least what is going in Noux itself:

```
[…]
[init -> noux] PID 0 -> SYSCALL FORK
[init -> noux] PID 0 -> SYSCALL WAIT4
[init -> noux] PID 5 -> SYSCALL STAT
[init -> noux] PID 5 -> SYSCALL EXECVE
[init -> noux] PID 5 -> SYSCALL STAT
[init -> noux] PID 5 -> SYSCALL GETTIMEOFDAY
[init -> noux] PID 5 -> SYSCALL STAT
[init -> noux] PID 5 -> SYSCALL OPEN
[init -> noux] PID 5 -> SYSCALL FTRUNCATE
[init -> noux] PID 5 -> SYSCALL FSTAT
[init -> noux] PID 5 -> SYSCALL GETTIMEOFDAY
[init -> noux] PID 5 -> SYSCALL FCNTL
[init -> noux] PID 5 -> SYSCALL WRITE
[init -> noux -> /bin/tar] DUMMY fstatat(): fstatat called, not implemented
[init -> noux] PID 5 -> SYSCALL FCNTL
[init -> noux] PID 5 -> SYSCALL FCNTL
[init -> noux] PID 5 -> SYSCALL WRITE
[init -> noux] PID 5 -> SYSCALL FCNTL
[init -> noux] PID 5 -> SYSCALL WRITE
[init -> noux] PID 5 -> SYSCALL GETTIMEOFDAY
[init -> noux] PID 5 -> SYSCALL CLOSE
[init -> noux] PID 5 -> SYSCALL FCNTL
[init -> noux] PID 5 -> SYSCALL WRITE
[init -> noux] PID 5 -> SYSCALL CLOSE
[init -> noux] child /bin/tar exited with exit value 2
[…]
```

The trace log was shortened to only contain the important information.

We now see at which point something went wrong. To be honest, we see the `DUMMY` message even without enabling the tracing of system calls. But there are situations where a application is actually stuck in a (blocking) system call and it is difficult to see in which.

Anyhow, `fstatat` is not properly implemented. At this point, we either have to add this function to the Genode's libc or rather add it to libc_noux. If we add it to the libc, not only applications running in Noux will benefit but all applications using the libc. Implementing it in libc_noux is the preferred way if there are special circumstances because we have to treat the function differently when used in Noux (e.g. `fork`).

For the sake of completeness here is a list of all files that were created by porting GNU tar to Genode's Noux runtime:

```
ports/ports/tar.hash
ports/ports/tar.port
ports/run/noux_tar.run
ports/src/noux-pkg/tar/target.mk
```

Extensive build rules example

The build rules for OpenSSH are much more extensive than the ones in the previous example. Let us take a quick look at those build rules to get a better understanding of possible challenges one may encounter while porting a program to Noux:

```
# This prefix 'magic' is needed because OpenSSH uses $exec_prefix
# while compiling (e.g. -DSSH_PATH) and in the end the $prefix and
# $exec_prefix path differ.

CONFIGURE_ARGS += --disable-ip6 \
                  […]
                  --exec-prefix= \
                  --bindir=/bin \
                  --sbindir=/bin \
                  --libexecdir=/bin
```

In addition to the normal configure options, we have to also define the path prefixes. The OpenSSH build system embeds certain paths in the ssh binary, which need to be changed for Noux.

```
INSTALL_TARGET  = install
```

Normally the Noux build rules (*noux.mk*) execute `make install-strip` to explicitly install binaries that are stripped of their debug symbols. The generated Makefile of OpenSSH does not use this target. It automatically strips the binaries when executing `make install`. Therefore, we set the variable `INSTALL_TARGET` to override the default behaviour of the Noux build rules.

```
LIBS += libcrypto libssl zlib libc_resolv
```

As OpenSSH depends on several libraries, we need to include these in the build Makefile. These libraries are runtime dependencies and need to be present when running OpenSSH in Noux.

Sometimes it is needed to patch the original build system. One way to do this is by applying a patch while preparing the source code. The other way is to do it before building the Noux package:

```
noux_built.tag: Makefile Makefile_patch

Makefile_patch: Makefile
        @#
        @# Our $(LDFLAGS) contain options which are usable by gcc(1)
        @# only. So instead of using ld(1) to link the binary, we have
        @# to use gcc(1).
        @#
        $(VERBOSE)sed -i 's|^LD=.*|LD=$(CC)|' Makefile
```

```
	@#
	@# We do not want to generate host-keys because we are crosscompiling
	@# and we can not run Genode binaries on the build system.
	@#
	$(VERBOSE)sed -i 's|^install:.*||' Makefile
	$(VERBOSE)sed -i 's|^install-nokeys:|install:|' Makefile
	@#
	@# The path of ssh(1) is hardcoded to $(bindir)/ssh which in our
	@# case is insufficient.
	@#
	$(VERBOSE)sed -i 's|^SSH_PROGRAM=.*|SSH_PROGRAM=/bin/ssh|' Makefile
```

The target *noux_built.tag* is a special target defined by the Noux build rules. It will be used by the build rules when building the Noux package. We add the `Makefile_patch` target as a dependency to it. So after configure is executed, the generated Makefile will be patched.

Autoconf's configure script checks if all requirements are fulfilled and therefore, tests if all required libraries are installed on the host system. This is done by linking a small test program against the particular library. Since these libraries are only build-time dependencies, we fool the configure script by providing dummy libraries:

```
#
# Make the zlib linking test succeed
#
Makefile: dummy_libs

LDFLAGS += -L$(PWD)

dummy_libs: libz.a libcrypto.a libssl.a

libcrypto.a:
	$(VERBOSE)$(AR) -rc $@
libssl.a:
	$(VERBOSE)$(AR) -rc $@
libz.a:
	$(VERBOSE)$(AR) -rc $@
```

© 2019 Genode Labs