Genode Tutorial

Author

Md Shifuddin Al Masud

MSc in Informatics (running)

Technical University of Munich

Introduction

Genode is an OS framework toolkit to build highly secure special purpose operating systems. Programs are developed to run on any of the operating systems. This framework lets the programs to communicate with each other and share resources but according to strict manner.

It is regularly used and developed on GNU/Linux. That's why it's the responsibility of the user (component developer) to ensure latest long term release (LTS).

Features

CPU architectures: Both x86 (32, 64 bit) and ARM.

Kernels: most members of the L4 family NOVA, Fiasco.OC, OKL4 v2.1, L4ka::Pistachio, L4/Fiasco, Linux, and a custom kernel

Virtualization: VirtualBox (on NOVA), L4Linux (on Fiasco.OC), and a custom runtime for Unix software

Get Started

To get Genode in your machine there are two options available. First one, get precompiled version and second one is to build Genode from source. I am describing the later alternative since building from source code repository is the best way to have up to date version.

Clone Source:

Simply clone the following git repository

Command:

git clone git://github.com/genodelabs/genode.git

Before starting further uses of source code let's discuss source tree structure.

Source-tree structure:

At the root of the directory tree, there are the following contents:

doc/: Documentation in plain text format.

tool/: Tools and scripts to support the build system, various boot loaders, the tool chain, and the management of 3rd-party source code.

repos/: The so-called source-code repositories, which contain the actual source code of the framework components.

os/: OS components such as the init component, device drivers, and basic system services.

libports/: Ports of popular open-source libraries, most importantly the C library. Among the 3rd-party libraries are Qt5, libSDL, freetype, Python, ncurses, Mesa, and libav2.2 are popular.

I will discuss source code building process in an exemplary way. Assume I want to build an OS and write a program/component for that OS and run that program on that OS too. So I have three part future task.

Build OS:

Build toolchain

Genode highly depends on custom tool chain for this purpose. First of all I need to prepare this tool chain.

Step 1: From <genode-dir> enter to tool.

Command:

cd tool

Step 2: Run ./tool_chain with specific CPU architecture. In our case I use arm.

Command:

./tool_chain arm

**Note: To observe available options you may run tool_chain without parameter

Command:

./tool_chain

Prepare base/kernel

OS relies on a base/kernel. Preparing the base of choice for OS is the essential part of getting full functional OS.

Step 1: Change directory to repos/ from <genode-dir>

Command:

cd repos

Step 2: Change directory to your base of choice. In our case base-foc.

Command:

cd base-foc

Step 3: Prepare the base

Command:

make prepare

Prepare Ports

We also need to prepare libports to be prepared during OS building session. Steps required to build libports..

Step 1: Change directory from <genode-dir> to libports/

Command:

cd libports

Step 2: Prepare libraries

Command:

make prepare

Create Build Directory

Here I will create a build directory from where, OS and components will be build. Build system never touches source tree. It will generate object files, drivers, programs in this dedicated build directory.

Step 1: Run the build command from the <genode-dir>

Command:

./tool/create_builddir foc_pbxa9

This command will create a build directory at build/foc_pbxa9 which is readily configured to use base-foc as base platform which I already prepared.

Change Configuration file

Step 1: Change the configuration file. Start from <genode-dir>, open build.conf file and insert MAKE += j4 and uncomment lines which are required to include libports directory.

Command:

cd build/foc-pbxa9/etc

Vi build.conf

Finally call the target.mk file from the OS. Before this change directory to build directory. In this example build/foc_pbxa9

Command:

make prepare

Create Components

Hello World

In this section of the tutorial I will show how to create and execute custom components. Historically always it starts with hello world. So our first example is a custom component which prints "Hello World" message.

Prepare directory structure

Step 1: Let's host a new source code directory in the repos/ directory. This custom repository holds source code and build rules for our hello world component as well as run script to run the component at genode.

Command:

cd <genode-dir>

mkdir repos/custom

Step 2: Create two subdirectories under repos/custom names src and run. Component source code and build rule resides in src directory where as run script in run directory.

Command:

cd <genode-dir>

mkdir repos/custom/src

mkdir repos/custom/run

Step 3: By convention, the *src/* directory contains further subdirectories for hosting different types of components, in particular *server* (services and protocol stacks), *driver* (hardware-device drivers), and *app* (applications). For the hello-world component, an appropriate location would be *src/app/hello/.*

Command:

mkdir -p repos/custom/src/app/hello

Write component & build description

Step 1: In this step I am working in the component directory that is hello/ . It holds source code and build description in this case target.mk. The main part of each component typically resides in a file called *main.cc*. Put following lines of code in main.cc

```
#include <base/printf.h>
int main()
{
Genode::printf("Hello world\n");
return 0;
}
```

Step 2: Put following build rules into target.mk file.

TARGET = hello

SRC_CC = main.cc

LIBS += base

Target represents component name, src_cc lists source files to be compiled and libs includes precompiled libraries to be used by this component. You can find out available libraries e.g from repos/base-foc/lib/mk directory.

Add component to build configuration

Let's move to build this newly created component. Change directory to foc_pbxa9 and build the component. Okay before this just add the component path to build.conf file. That is I need to extend build configuration. So add the following line to <genode-dir>/build/foc_pbxa9/etc/build.conf

REPOSITORIES += $(GENODE_DIR)/repos/custom.

Build component

Command:

make app/hello

Create Run Script

We are at the edge to run our first component. Now I require a run script as I mentioned earlier. Write a run script at <genode-dir>/repos/custom/run/hello.run file. Add the following lines to that file.

```
build "core init app/hello"
create_boot_directory
install_config {
<config>
<parent-provides>
<service name="LOG"/>
<service name="RM"/>
</parent-provides>
<default-route>
<any-service> <parent/> </any-service>
</default-route>
<start name="hello">
<resource name="RAM" quantum="10M"/>
</start>
</config>
}
build_boot_image "core init custom"
append qemu_args "-nographic -m 64"
run_genode_until {child "hello" exited with exit value 0} 10
```

This run script performs the following steps:

1. It builds the components core, init, and app/custom.
2. It creates a fresh boot directory at <*build-dir*>*/var/run/custom*. This directory contains all files that will end up in the final boot image.
3. It creates a configuration for the init component.
4. It assembles a boot image with the executable ELF binaries core, init, and custom.
5. It instructs Qemu (if used) to disable the graphical output.

6. It triggers the execution of the system scenario and watches the log output for the given regular expression. The execution ends when the log output appears or after a timeout of 10 seconds.

Run Component

The run script can be executed within the build directory via the command:

make run/hello

Output should look like..

[init -> hello] Hello world
[init] virtual void Genode::Child_policy::exit(int):
child "hello" exited with exit value 0
Run script execution successful.

Create Threads and Run

In this example I am going to show how to create multiple threads and execute them. Objectives are mentioned below according to precedence.

1. Create several threads.
2. Set precedence of created threads
3. Print precedence graph
4. Run threads according to precedence

Create Threads

In Genode framework creating threads has little hurdle. All you need to extends Thread_base class and create a object of that subclass. This subclass inherits various virtual and pure virtual methods from Thread_base.

//sample of the subclass which extends Thread_base

// Main example is more complex

#include <base/thread.h>

Class Gthread:public Thread_base

{

Public:

```
        // constructor
        GThread (size_t Iight, const char *name)
        :Thread_base(Iight, name, THREAD_STACK_SIZE, Type::NORMAL)
        {

        }


        // pure virtual of Thread_base

        // I create a definition of that

        Public void entry()

        {

                …

                ...

        }

        …

        ...

};

 // Create Thread in main.cc
 GThread thread0(200, "thread one");
 GThread thread1(300, "thread two");
 GThread thread2(200, "thread three");
```

Set Precedence of created threads

I used struct to hold the threads and their precedence relations in a link list way. This struct has three members. First one is thread itself then thread name and finally I have pointer which points next thread.

```
// struct to hold threads and their precedence relation
typedef struct thread_node
{
    GThread thread;
    const char* thread_name;
```

```
    thread_node *next;
}thread_node;
```

Next I create objects of that struct to obtain our desired precedence.

```
// create thread_node with precedence
thread_node third_node = {thread2, "thread three", NULL};
thread_node second_node = {thread1,"thread two",&third_node};
thread_node first_node  = {thread0, "thread one",&second_node};
```

Print Precedence Graph

Here I show how threads are connected one with another.

```
// function to show precedence relation
void print_precedence_graph(thread_node *node)
{
    Genode::printf("Thread precedence.....\n");

    while(node != NULL)
    {
        Genode::printf("%s\n", node->thread_name);
        node = node->next;
    }

     Genode::printf(".....\n");

}

//call print_precedence_graph function with the address of first_node in main.cc

Print_precedence_graph(&first_node);
```

Run threads according to precedence:

I iterate through the thread_node link list and run each thread by invoking start() method. Thread start() method invokes entry() method for each and every thread.

```
// function to start threads
void start_threads(thread_node *node)
{
```

```
    while (node != NULL)
    {
        node->thread.start();
        node = node->next;
    }
}
```

//call start_threads method with the address of first node in main.cc.

```
start_threads(&first_node)
```

Automotive Example

This is second consecutive example and an interesting one. Here I create two threads and assign critical and noncritical operations respectively. Finally run the threads with different input and show which input is valid and which one is invalid. Creation and start of thread remain same as previous example. That's why it's not discussed here. Only new aspects and ideas are elaborated in this example. For this example below are the objectives.

1. Assign operations to threads.
2. Run thread with different operations.

Assign operations

I have created two threads named critical and noncritical. Operations are also divided into two subdivision named critical operation and noncritical operation. Then assigned critical operations to critical thread and noncritical operations to noncritical thread. Below are some examples of critical and noncritical operations..

```
// ops[0]: Critical operations
 ops_count[0] = 4;

 ops[0][0] = (Op){ 5, "ENABLE_ALL_WHEEL_DRIVE" };
 ops[0][1] = (Op){ 6, "DISABLE_ALL_WHEEL_DRIVE" };
 ops[0][2] = (Op){ 7, "ENABLE_ELECTRONIC_STABILITY_CONTROL" };
 ops[0][3] = (Op){ 8, "DISABLE_ELECTRONIC_STABILITY_CONTROL" };

// ops[1]: noncritical operations
ops_count[1] = 4;
```

```
ops[1][0] = (Op){ 10, "ENABLE_WIRELESS_LAN" };
ops[1][1] = (Op){ 11, "DISABLE_WIRELESS_LAN" };
ops[1][2] = (Op){ 12, "ENABLE_TRAFFIC_LIGHT_FEEDBACK" };
ops[1][3] = (Op){ 13, "DISABLE_TRAFFIC_LIGHT_FEEDBACK" }
```

Run thread with different operations

In this phase I send two operations to thread0 which is a critical thread and start the thread. Corresponding thread entry method checks whether these two operations are valid or not. If the operation lies under critical operation it prints valid operation and if not invalid operation.

```
// critical thread

umword_t data0[2] = {6,8};
 thread0.send_operation(data0,2);
 thread0.set_current(0);
 thread0.start();
```

Same happens for thread1 which is a non critical thread. I sent three operations and among them two are valid for this noncritical thread and one is invalid.

```
//noncritical thread

umword_t data1[3] = {10, 12, 5};
 thread1.send_operation(data1,3);
 thread1.set_current(1);
```

Observations of Genode:

In some cases, genode misses one or more threads among multiple created threads. We need to keep it in our considerations when working with threads in genode.