Scratchbox 2: Internals and Architecture

[for software versions 2.0 – 2.2]

Lauri T. Aarnio

1st Edition, April 25, 2012.

Table of Contents

Preface to the first (public) edition

This document was originally written for developers of software development kits (SDKs). It was mainly targeted to our internal audience at Nokia, while an alternative Scratchbox 2 -based SDK was being developed for the Maemo devices.

Our intention was to release this document to the Scratchbox 2 community at the same time with the final release of Maemo SDK+. However, in February 2011 priorities of many things changed, especially things which were related to Maemo/MeeGo operating systems. One side-effect was that the development of the SDK+ stalled.

But Scratchbox 2 itself has always been independent of the target environments. It is being used as a platform for development environments for several Linux-based systems, like Tizen, the Mer project, the Raspberry Pi, and WebOS, to name a few.

Also, we have used Scratchbox 2 as a platform for analyzing SW package build process behavior. For example, the latest development versions can create process call hierarchy graphs, and can be used to alter network addresses as well as create alternative file system namespaces. Scratchbox 2 has been proven to be a versatile tool for solving certain type of environment virtualization problems with Linux.

This is the first public edition of *Scratchbox 2 Internals and Architecture*. It still refers mostly to software versions 2.0 – 2.2.

At the time of writing this preface, 2.2.4 is the latest stable version of Scratchbox 2, but the next stable major release, 2.4, is not necessarily too far in the future. I have added some short notes *[written in italics]* to places where there are notable changes in the development branch: The latest development version, 2.3.52, contains some new features, and some old ones have been rewritten for better performance and scalability.

Lauri T. Aarnio
lauri.t.aarnio@nokia.com
Helsinki, Finland

Foreword and Introduction to Scratchbox 2

Documenting a system as complex as Scratchbox 2 is not an easy task. Lauri Aarnio has demonstrated amazing endurance in coming up with ways to express the complicated inner workings, this process has also served in clarifying even to ourselves the reasons and the true nature of many of the mapping engine peculiarities. This paper is not an easy read by any means, and it cannot be really recommended as a beginners guide to using Scratchbox 2, but it is definitely invaluable for anyone who wants to understand it, and hopefully serves well those who wish to contribute to it.

Scratchbox 2 is a continuation of Scratchbox 1 in terms of approach to solving cross-compilation challenges, even if it isn't 100% compatible and differs greatly in the implementation. The topic of cross-compilation is convoluted. Looking at the most basic scenario, where one is using a compiler that produces output to a different hardware architecture than the one it's running on, it becomes primarily a task of making sure the right compiler is being used to build all of the code, and that it finds headers and libraries that belong to the target system instead of the host.

Elaborating this further, various build systems have configuration settings for the program being built, sometimes these are hard-coded in the Makefiles, sometimes taken from environment variables or using some utility to detect things about the build or target environment. Especially the very popular open source configuration tools under the Autotools umbrella make heavy use of automatically detecting library versions, function existence, paths to utilities and their versions, include and library paths etc. Autoconf compiles small test snippets and sometimes even executes the resulting binaries while detecting target system capabilities.

Next level of escalation is then managing the target build environment on the host machine. For Maemo this means implementing a mechanism for installing and updating target components from the debian repositories and also making sure the host system has build tools that are compatible with what the target system requires.

All of these need to be tackled in order to have a functioning development environment. Scratchbox 2 does this by disguising the host system so that it appears as the target system, while retaining the ability to execute the host tools. This dualism requires quite substantial workarounds at times to control where each file is accessed from in order to maintain the illusion of being the target, while providing the functionality of the host.

Lauri Leukkunen
Founder of the Scratchbox2 project

1. Scratchbox 2, a Cross-Compilation Toolkit

Scratchbox 2 is a non-traditional solution to the cross-compilation problem:

Scratchbox 2 is used to create a virtual, native-like development environment. In other words, it makes a development machine look like a virtual target system to the software development tools.

The design of Scratchbox 2 is based on the following principles:

- Flexibility: The system is highly configurable. Rulesets can be used to define the level and nature of the virtualization.
- User experience: The virtual development environment should let the user access “best of both worlds” with minimal disturbance: The virtual environment should resemble the target system as closely as possible, and the user should also have seamless access to the development host.
- Performance: Scratchbox 2 should have minimal effect on performance. Usually virtualization introduces some performance penalties, and so does Scratchbox 2. But the design enables use of the fastest possible tools whenever possible: Tools that are architecture-neutral can be executed as host-architecture-compatible binaries, while binaries which depend on being executed on the target system will find themselves running on a target-like environment, engaging target CPU emulation only if needed. This is completely transparent to the user and to the build process.
- Security: Scratchbox 2 does not need any special privileges or kernel-level services; it is completely based on features that are available for all ordinary users in Linux systems.
- Concurrency: A single user may have several sessions open at the same time (each session may or may not share configuration and data with the other sessions), and naturally several users can use Scratchbox 2 on the same host without interfering each other.

Scratchbox 2 works on the Linux OS; both the host and the target are currently expected to be Linux systems. The host computer can be an ordinary Linux PC (32- or 64-bit intel-based architecture), while the supported target architectures include ARM, PPC, MIPS, x86 and SH.

It should be noted that Scratchbox 2 is not an SDK (software development kit) itself. Instead, it is a cross-compilation toolkit; an engine which can be used to build SDKs.

1.1 Related components: Other building blocks for a complete SDK

An SDK typically consists of several parts in addition to Scratchbox 2 (fig 1):

- First, a cross compiler and related tools (assembler, linker, etc.) are needed.
- Second, a target file system containing all target-specific libraries, header files, etc. is mandatory. This is nicknamed “the rootstrap”, as it is usually a copy of target device's root filesystem.
- Third, usually a set of build tools is also needed. A common mistake is to think that a cross-

compiler would be enough to create binary programs for the target; this is not the case especially with existing OSS (Open Source Software) projects. Typically a set of ca. 100 other tools are needed for successful building of an OSS project (See fig. 2. for an example)

- Fourth, a target hardware emulator is needed, if the target CPU architecture is not directly compatible with the host CPU. *Qemu,* an open source hardware emulator is typically used for this task. *Qemu* is used in so-called “user mode”, which only emulates the CPU. Target hardware or complete system is *not* emulated, instead it translates target's system calls to calls to host system's kernel.
- An IDE is typically added to be used as the user interface.
- Also, the complete SDK needs administrative tools: Tools for installing it, maintaining versions of the above components, etc.

Scratchbox 2 is the “glue” which binds all of the above components together and creates a working, virtual development environment.

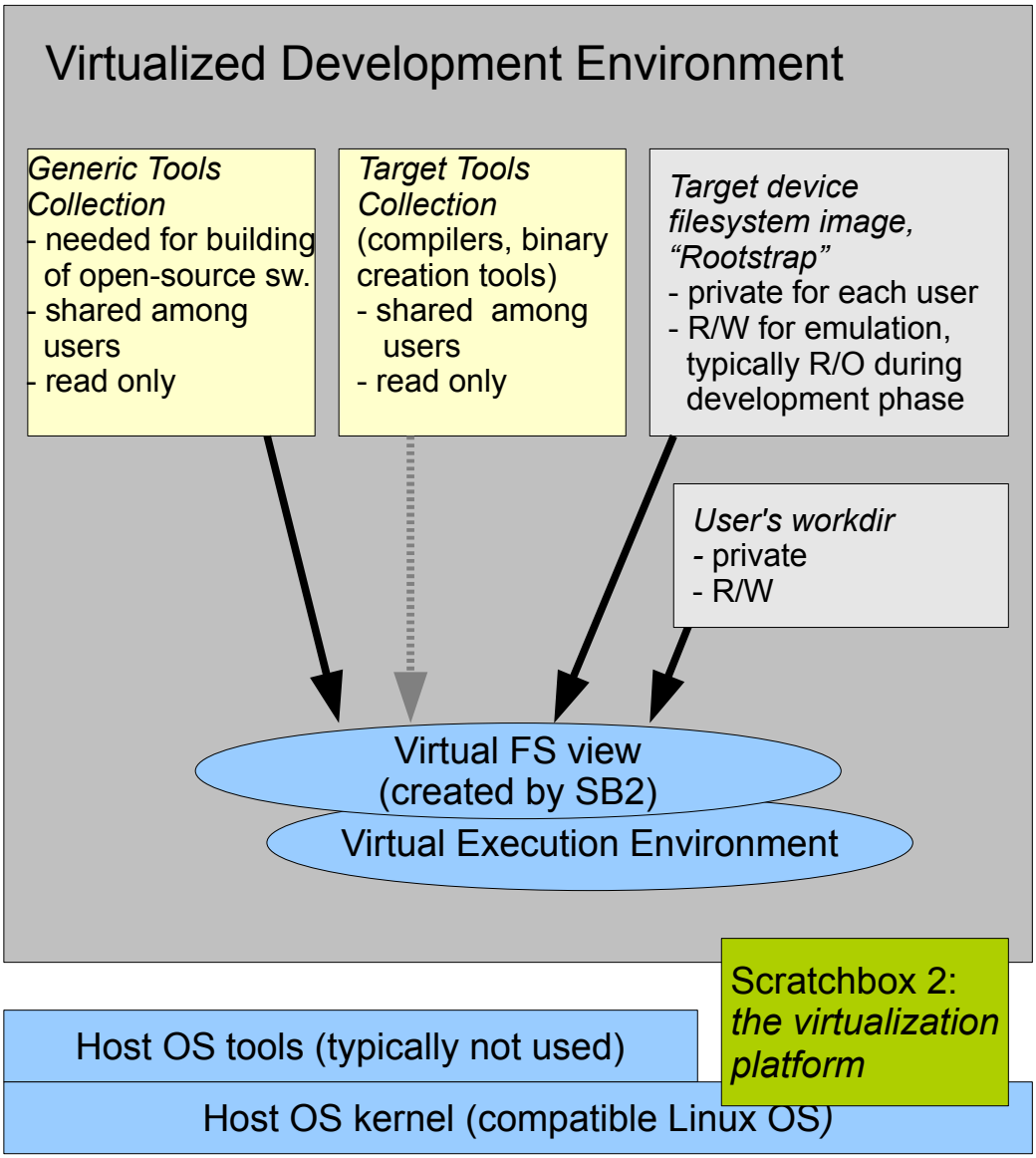

Fig. 1. Scratchbox 2 as an SDK core engine

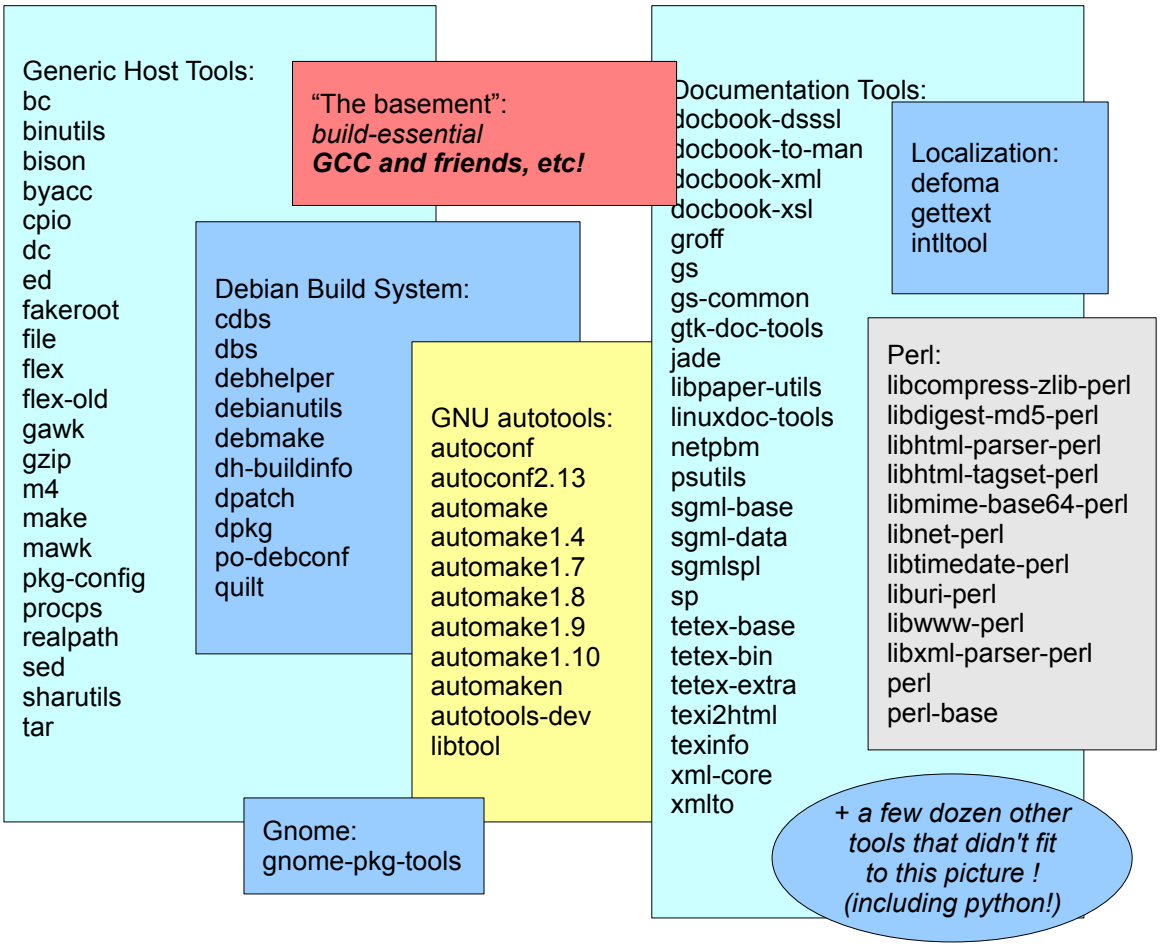

Fig.2. Typical set of tools for a Debian Linux based SDK.

1.2 A bit of history: Scratchbox 1 vs. Scratchbox 2

Scratchbox 2 is not a direct replacement of Scratchbox 1, it can be used to fulfill the same function, but Scratchbox 2 provides a much more flexible and easier to maintain implementation.

As an example, a minimal Scratchbox 2 installation

- takes less than a megabyte (minimal Scratchbox 1 is around 100 megabytes),
- does not require super user rights (e.g. does not depend on kernel's *binfmt_misc* feature, no need to mount (or bind mount) directories, and does not use *chroot* system calls – Scratchbox 1 requires all of those)
- and out of the box supports multiple simultaneous sessions for different target configurations. providing basic cross-compilation support.

A more complete development environment requires a controlled set of build tools, which Scratchbox 2 supports independently of the host machine's distribution by letting the user to specify a directory from which all binaries are used and executed from.

2. Operation of Scratchbox 2

Operation of Scratchbox 2 is heavily based on applying rules, which define the composition of the virtual environment. There are two types of rules: File system rules are used to define how the file system appears to the applications, and execution rules can be used to alter the way how applications are started. *[version 2.3 has added a third set of rules for limited network address translation capabilities]*

Several ready-made rulesets are distributed together with the software, so that the user usually doesn't have to modify or add any rules. However, an SDK which is based on Scratchbox 2 may include additional rulesets for special purposes – the system is easily configurable and relatively easy to maintain without making changes to the software itself.

Scratchbox 2 has been designed to operate completely as an user-level solution: That is, no special permission or special kernel-level services are required when executing programs in Scratchbox 2. It is even possible to install it so that no protected directories or files are affected.

This chapter will present an overview of the Scratchbox 2 environment from the user point of view. Chapters 3, 4 and 5 contain slightly more detailed view of configuration, interfaces and internals.

2.1 Targets and Sessions: User-level virtualization in practice

The two most important concepts are *target* and *session*[1].

A *target* is simply a symbolic name for a configuration set. A *target* does not contain anything that is active, like running processes; hence a user never "works inside a target". Instead, sessions are used for all active operations - always, even if the users don't notice them[2]. In the simplest case, the *target* contains everything that is needed to create a *session.*

In practice, the *target* contains configuration for the cross-compiler, location of the target root file system, cpu transparency method and the tools collection. These settings are only read when a session is created. After that all configuration information comes from the session.

Naturally, it is possible to have multiple sessions open in parallel, all based on the same target, all sharing more or less from the configuration which was initially read from the target settings. Some noteworthy things about sessions are that

- The target root file system is usually shared between sessions that were created from the same target. If contents of it are changed from one session, the changes are visible in the other sessions also (but this is not a rule:There are some exceptions; private, temporary target roots can also be used)
- The */tmp* directory is usually private to each session.

The session's configuration determines how the development environment is presented to the user: A virtual file system view[3] is created, and also the execution of other programs is enhanced: Binaries for the target system can be executed on the host computer. The principle is presented in Fig.3.

1 A note for users who are familiar with the older Scratchbox 1; *"targets"* were also used with that, but here the definition is slightly different. And a *session* is a new concept; somewhat similar to the "login shell" of SB1, but SB2's sessions are more flexible.

2 It is also possible to explicitly create sessions for more complex use cases: those are called "persistent sessions".

3 Scratchbox2 creates *virtual file system views*, it does not introduce any real virtual filesystems. The former can be done as an user-level service, while the latter usually requires kernel-level support (remember that the Scratchbox 2 does not require or introduce any special kernel-level services)

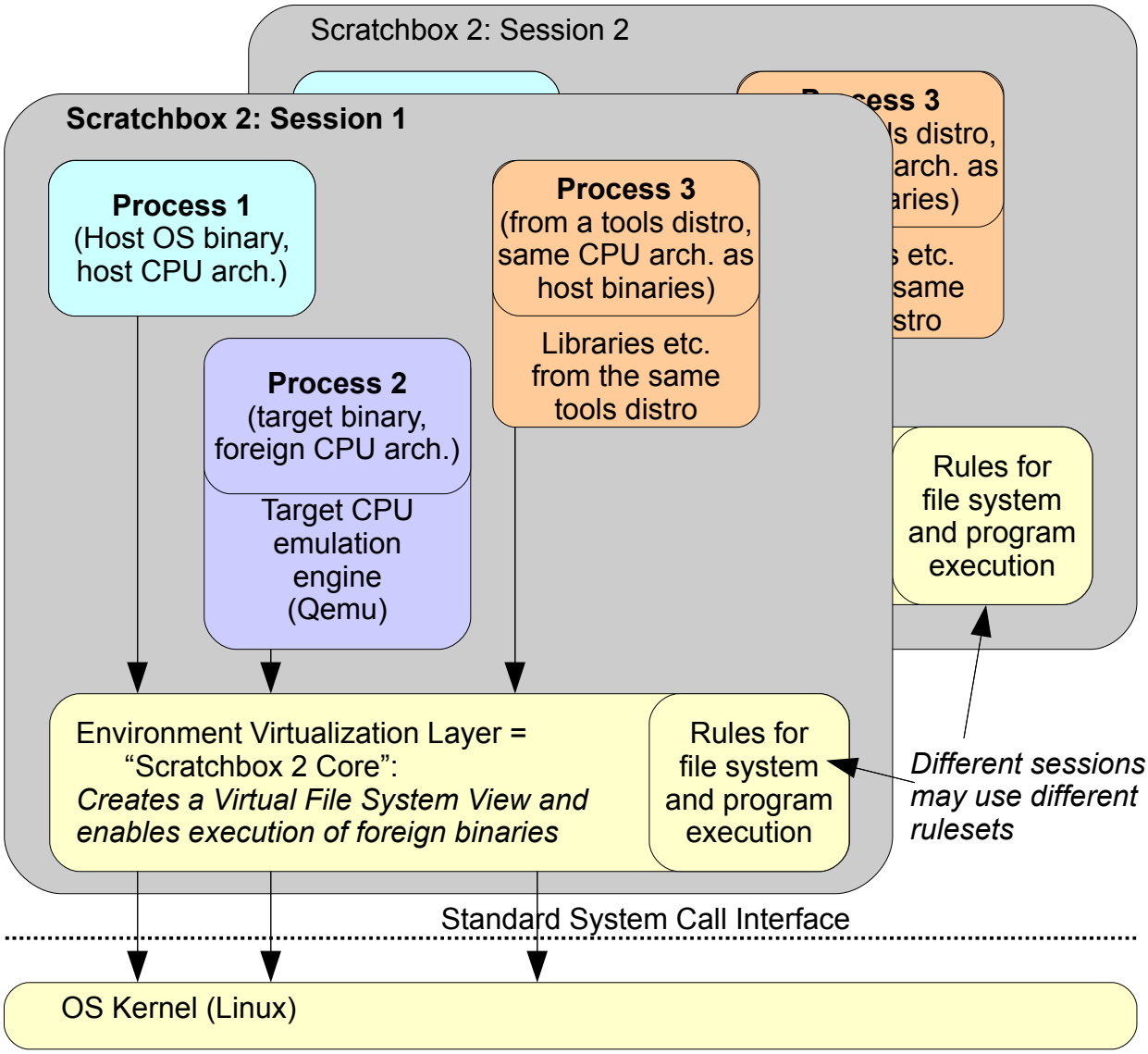

Fig.3. Sessions and execution of foreign binaries inside the session.

Three kind of processes can be seen in fig.3: Host binaries, incompatible target binaries and architecturally compatible, but environmentally incompatible binaries. Scratchbox 2 takes care of execution of all these cases transparently (there is more about program execution in ch. 2.4)

2.2 Typical workflow and the various mapping modes

Scratchbox 2 enforces a practice where software development and execution environments are separated. Especially for embedded devices these two have very different requirements: It is quite important to understand that all components that are present in the development environment (see fig.2) are not present in the actual execution environment (the device), and they should not be present in the emulated execution environment, either.

Fig.4 presents a typical workflow in the Scratchbox 2-created cross-compilation environment. The first two tasks (developing and building) are executed in a *development environment*, while installation and testing happen in the *execution environment.*

Scratchbox 2 creates different enviroments by using different rulesets (compare with fig.3). The rulesets are called *mapping modes*, as they determine how the files and processes are *mapped* to the virtualized environment.

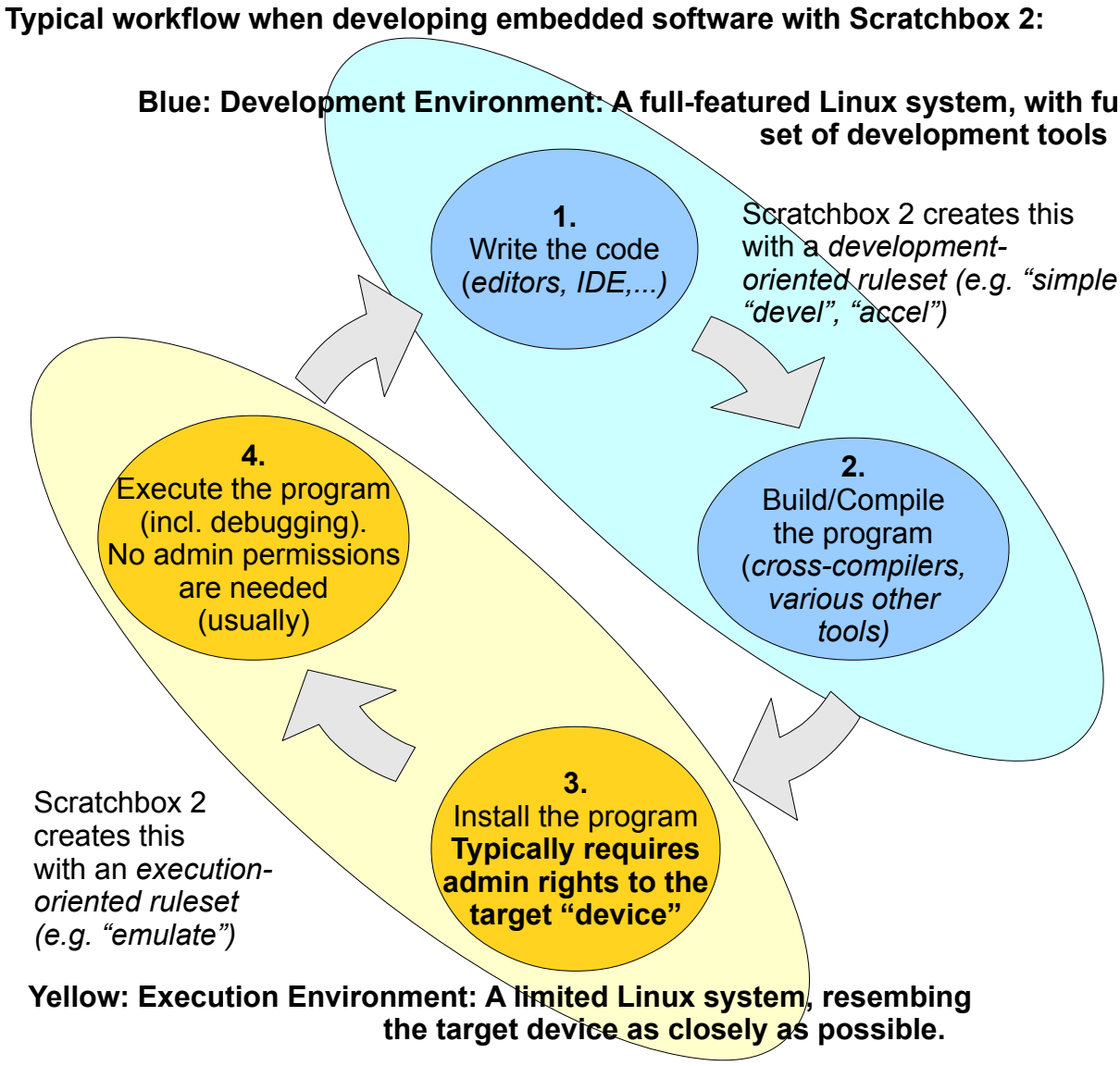

Fig.4. A typical software developer's workflow.

The distribution includes mapping modes for both the development and execution use cases: **emulate** is the target system emulation mode which should be used when installing programs to the target root file system and executing them (yellow area in fig.4). For development (blue side of fig.4), three modes are available: **simple**[4]**, devel**[5] and **accel**[6]. *[devel has been removed from version 2.3; instead, the name **devel** is an alias and activates the **accel** mode]*

Targets are usually created so that one of the development modes is the default mapping mode, but a session can also be created so that it uses another mapping mode. For example, the *emulate* mode needs to be explicitly activated when the "yellow tasks" of fig.4 are needed.

Although it is possible that the users (or the SDK developers) may introduce their own rulesets, the intention is that the ready-made rulesets that are distributed with Scratchbox 2 should be enough for most purposes and no customization should be necessary.

2.3 Processing of pathnames ("mapping" of files)

Scratchbox 2 builds the virtual file system by changing the paths to files before the paths are given to the operating system. Probably the most typical conversion is that a *path prefix* is added to the path which was provided by the application.

For example, when an application calls *open()* to open file */etc/timezone*, Scratchbox 2 captures the parameters and changes the path to point to different destination. Typically, the mapping rules may cause it to be mapped to the target root: A prefix will be added to the path, and the real file which will be opened might be */home/user/rootstrap/etc/timezone*[7]. This happens inside the Scratchbox 2 core, so the process which called *open()* does not notice anything.

Fig.5 presents a simplified example: A development-oriented mapping mode. Five rules, represented by red arrows, can be seen in the picture: Two *path prefixes* are mapped to the tools (*/bin* and */usr/bin*) and two to the target root (*/lib* and */usr*). Additionally, home directories are not mapped at all; */home* from the host is also available in the virtual FS view.

Fig 6. contains a setup where mapping has been configured to create an emulated target environment. In that case almost everything will be mapped to the target root.

In practice, about a dozen rules is the minimum that is needed to create a working environment. A few more, and the environment can be said to be even a comfortable place to work[8] – even more comfortable than what a *chroot*-based development environment could ever be. However, there are tools that require more than what was possible with the basic setup - for example, a configuration file which belongs to a tool might need to be mapped with a separate rule to the "tools" directory, even if every other file in the same directory will be mapped to the target root. The result was that the "devel" mode, which tries to support a fairly large set of tools (see fig.2) now contains a few dozen rules.

---

4 *simple* supports the most common development tools, but does not cover more complex cases (e.g. some GUI-releated tools or building extensions for scripting languages like perl and python can not be done in the simple mode). Most tools come from the host OS environment in this mode.

5 *devel* is a more complete development mode, supporting a larger set of development tools, and also some backward-compatibility rules for compiling packages that have unnecessary dependencies to the Scratchbox 1 environment. The downside is that it is also a lot more complex. This is mostly intended to be used when developing for a debian-based Linux distribution, with a separate toolset (which can be taken from e.g. Debian Linux)

6 *accel* is a *devel*-like mode, but assumes that the development tools have been compiled from the same sources as what the target file system contains (e.g. target might consist of ARM binaries, tools consists of x86 binaries of exactly the same programs). This has proven to be a better solution than what could be done with the *devel* mode.

7 The exact location of the target root file system is of course a configurable item. */home/user/rootstrap* is used here just to keep the examples simple and easy to read.

8 This is what the "simple" mapping mode is trying to accomplish.

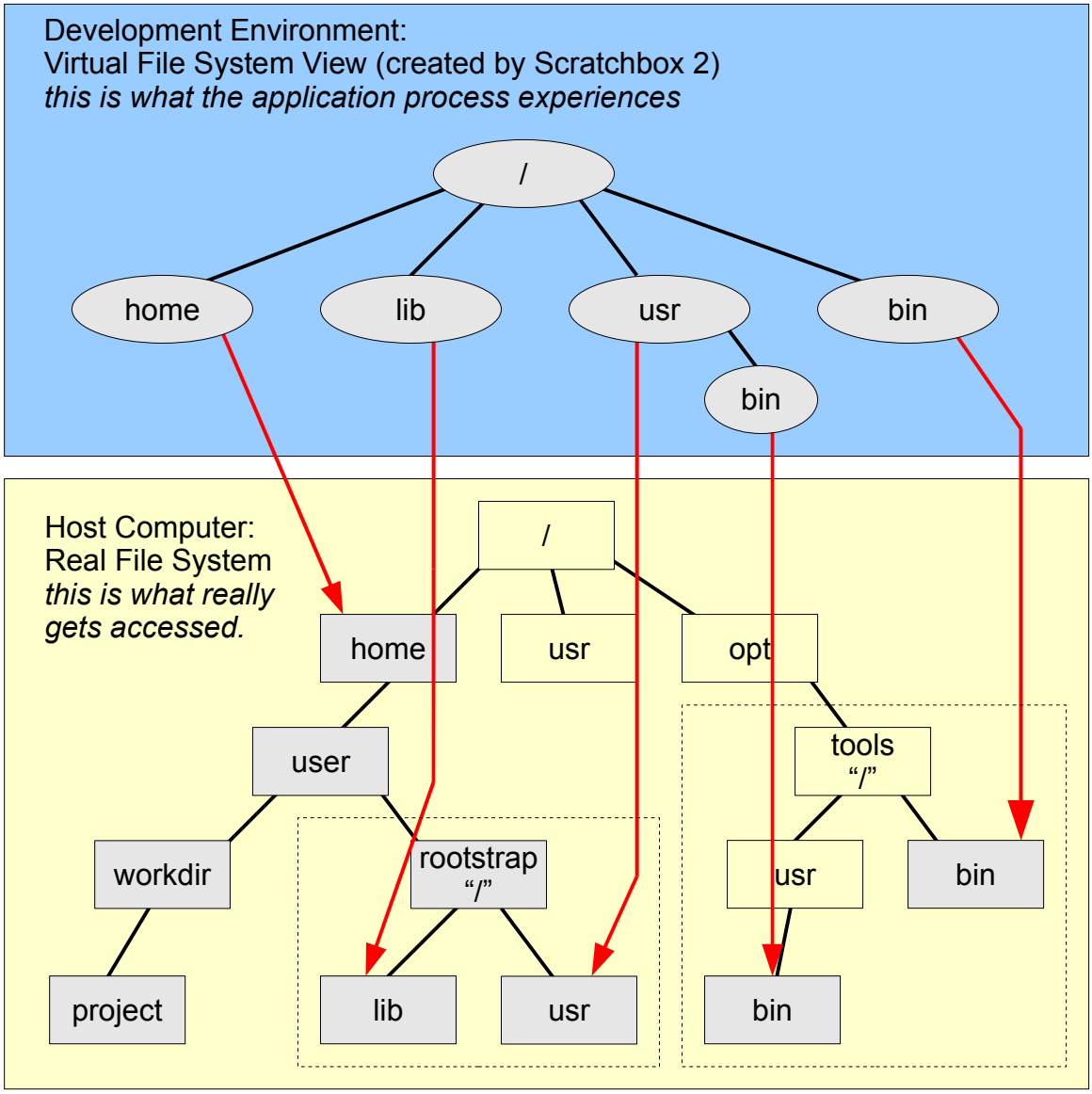

Fig.5. Mapping of pathnames (development-oriented mode, simplified example)

Scratchbox 2 can do mapping to both directions: In most cases *forward* mapping is used (e.g. the red arrows in figures 5 and 6 represent forward mapping), but there are cases where *reverse mapping* is performed:

For example, if the application uses the *chdir()* function to change the current directory to */usr/bin*, the mapping engine might change the path to either */opt/tools/usr/bin* (if a development-oriented ruleset is active: fig.5) or */home/user/rootstrap/usr/bin* (device emulation active: fig.6) and relay that path to the operating system. However, In the opposite case, when the application then wants to get the current directory with e.g. *getcwd()* function, the system first reports the real path (*/opt/tools/usr/bin* or */home/user/rootstrap/usr/bin)* to the Scratchbox 2 mapping engine, which reverses the mapping and returns */usr/bin* to the application.

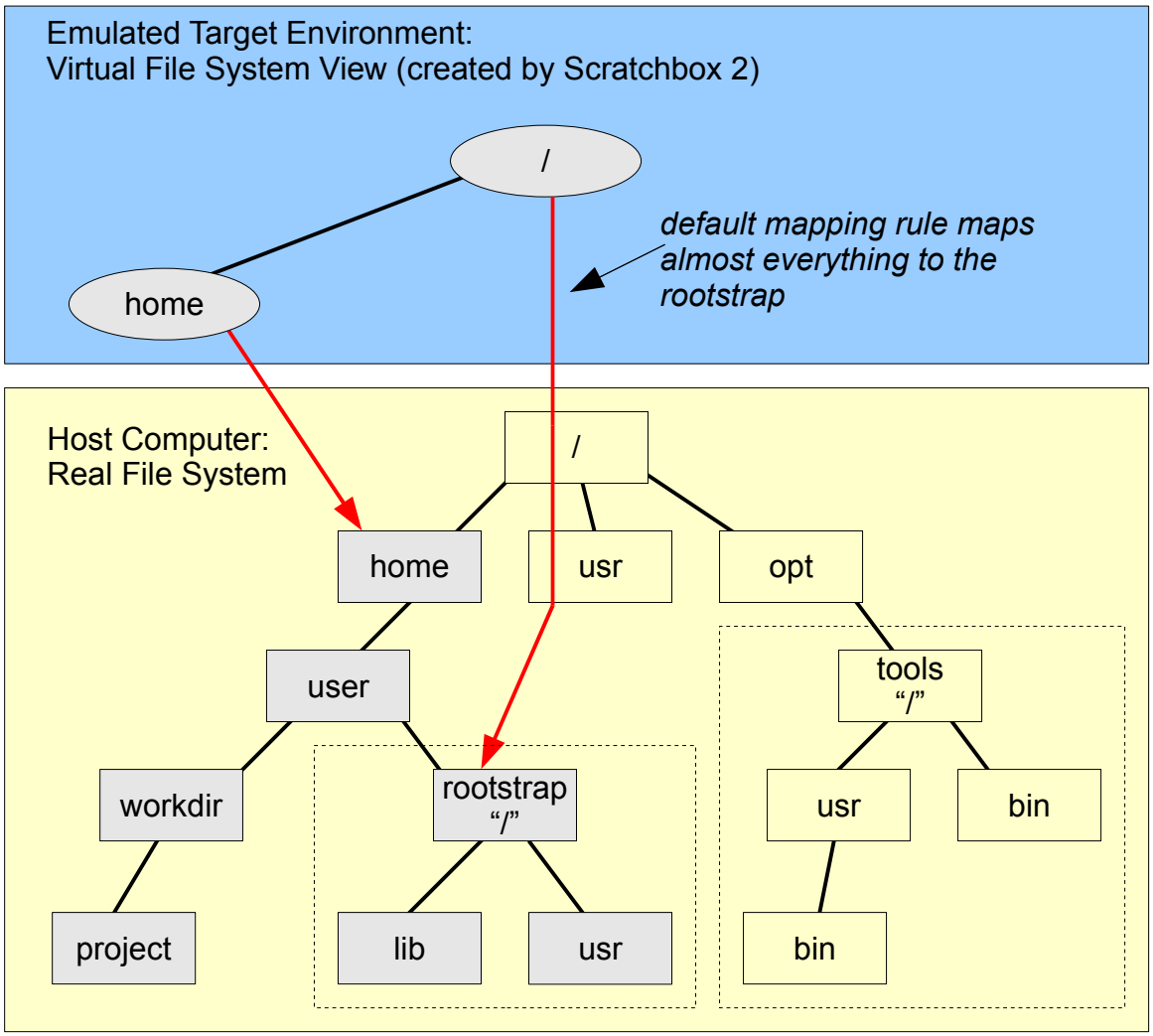

Fig.6. Mapping of pathnames (device emulation mode, simplified example)

Also, one extremely important detail that is worth noticing is the handling of symbolic links.

Suppose that the user has created a symbolic link to the "project" directory: If */home/user/workdir/project/bake* is a symbolic link to */usr/bin/make*, then what happens when the user executes "*bake*" from the project directory completely depends on the mapping mode:

1. In the development-oriented mode (fig.5), Scratchbox 2 interprets the link and maps references to */home/user/workdir/project/bake* to */opt/tools/usr/bin/make.*
2. In the target emulation mode (fig.6), Scratchbox 2 interprets the link and maps it to */home/user/rootstrap/usr/bin/make*.
3. If the user examines the link outside of the session, the link of course points to the system's real */usr/bin/make*.

In general, handling of symbolic links and other path-related algorithms are explained in the *path_resolution* manual page of Linux. Scratchbox 2 follows the guidelines explained there while performing the path mapping operations, so that file system access semantics are preserved.

2.4 Execution of programs

For host-compatible binaries, Scratchbox 2 is completely transparent from the process point of view: “Ordinary” programs don't need any kind of special treatment (like recompilation or preprocessing) to work in the Scratchbox 2 environment. It uses a *library preloading* mechanism internally, which means that references to symbols that normally belong to the operating system's libraries can be redirected to the Scratchbox 2's *preload library* for additional processing. The result is that Scratchbox 2 is directly compatible with almost all binary programs.

This mechanism requires that the executable is a dynamically linked binary and not a statically linked one. Fortunately dynamic linking is the default in Linux, and there are only a few exceptions where static linking is needed.

See fig.7. for an example: It presents how an architecturally compatible, dynamically linked binary can be executed either inside the session or completely outside of Scratchbox 2.

There are currently two ways how statically linked binaries can be handled: First, special wrappers have been developed for the most common cases where a statically linked binary is used, and Scratchbox 2 will log a warning message if/when such programs are executed [this was the only option for versions 2.2 – 2.2.3]

Newer versions have an alternative: Version 2.2.4 *[also, 2.3.4+]* support execution of statically linked host's binaries by running them with a second Qemu[9], using same principles which are applied to incompatible target CPU architectures.

For incompatible CPU architectures, i.e. if the target architecture is not compatible with the host architecture, the situation becomes slightly more complex: A CPU emulator must be used. This is presented in fig. 8.

Use of the CPU emulator is also transparent to the user and to the running processes: Scratchbox 2 extends the way how programs are started by first examining the program that will be executed and applying rules that might change or modify parameters before the actual *exec*-operation is performed. For example,

- Host OS binaries are easy: They see the virtual file system view, but otherwise everything tends to be similar to the host OS environment.
- Binaries, which are compatible with the host CPU but are not compiled for the host Linux distribution need to use special startup trick, so that it is possible to specify a non-standard location of the dynamic libraries that will be used. This applies to a situation where the target system uses same CPU architecture as the host, as well as setups where the cross-compilation tools are used from a different Linux distribution than what is used on the host.
- The third case concerns execution of incompatible binaries, i.e. when a CPU emulator is needed (fig.8).

The result is that when a program needs to be executed, everything just works “magically” - even between different kinds of executables inside Scratchbox 2 : For example, binaries compiled for the x86 architecture can use the standard *execl()*[10] function to start a binary which has been compiled for the ARM target.

9 This feature was developed at Samsung – many thanks to Rafał, Mike and Karol!

10 or one of the other six *exec* variants.

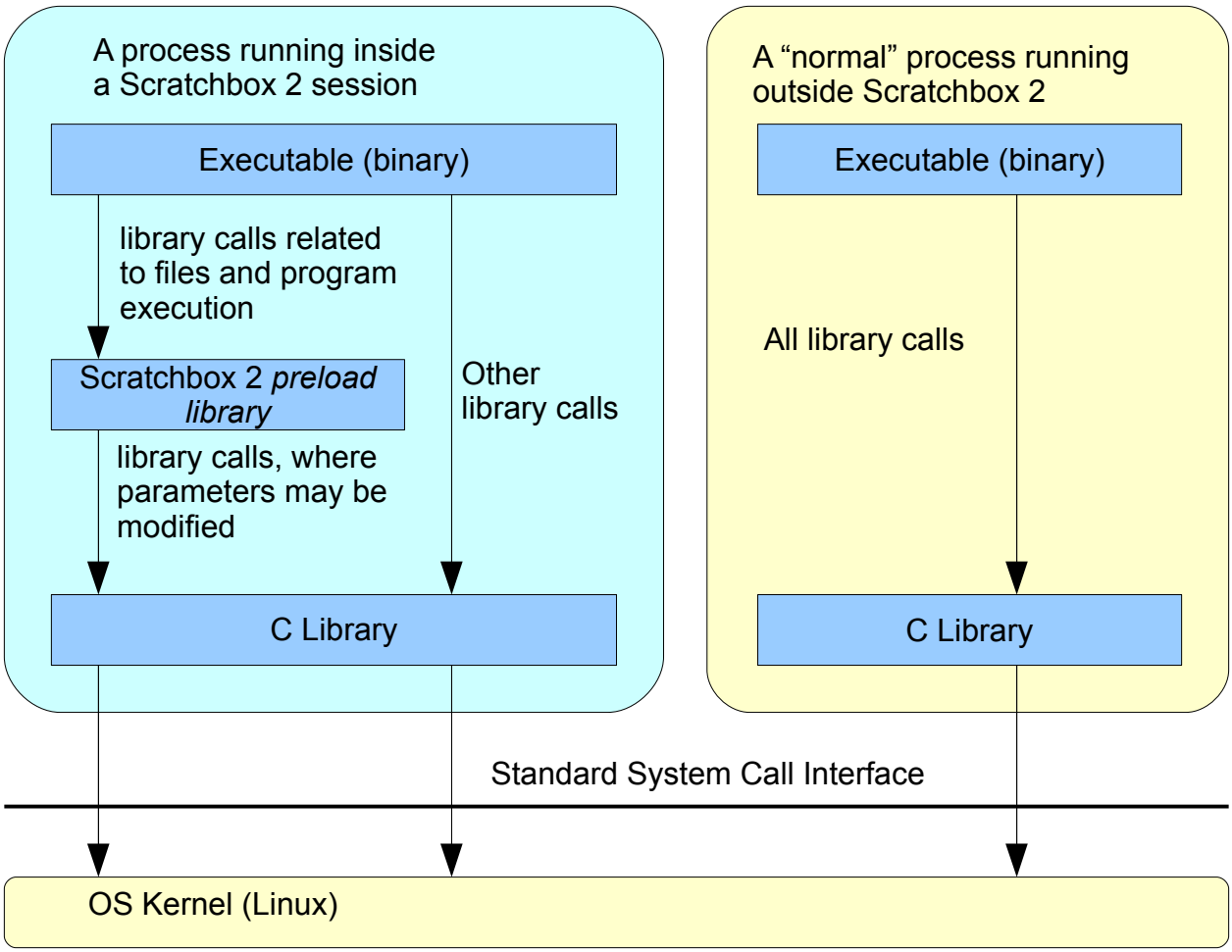

Fig.7. Execution of host-architecture-compatible binaries

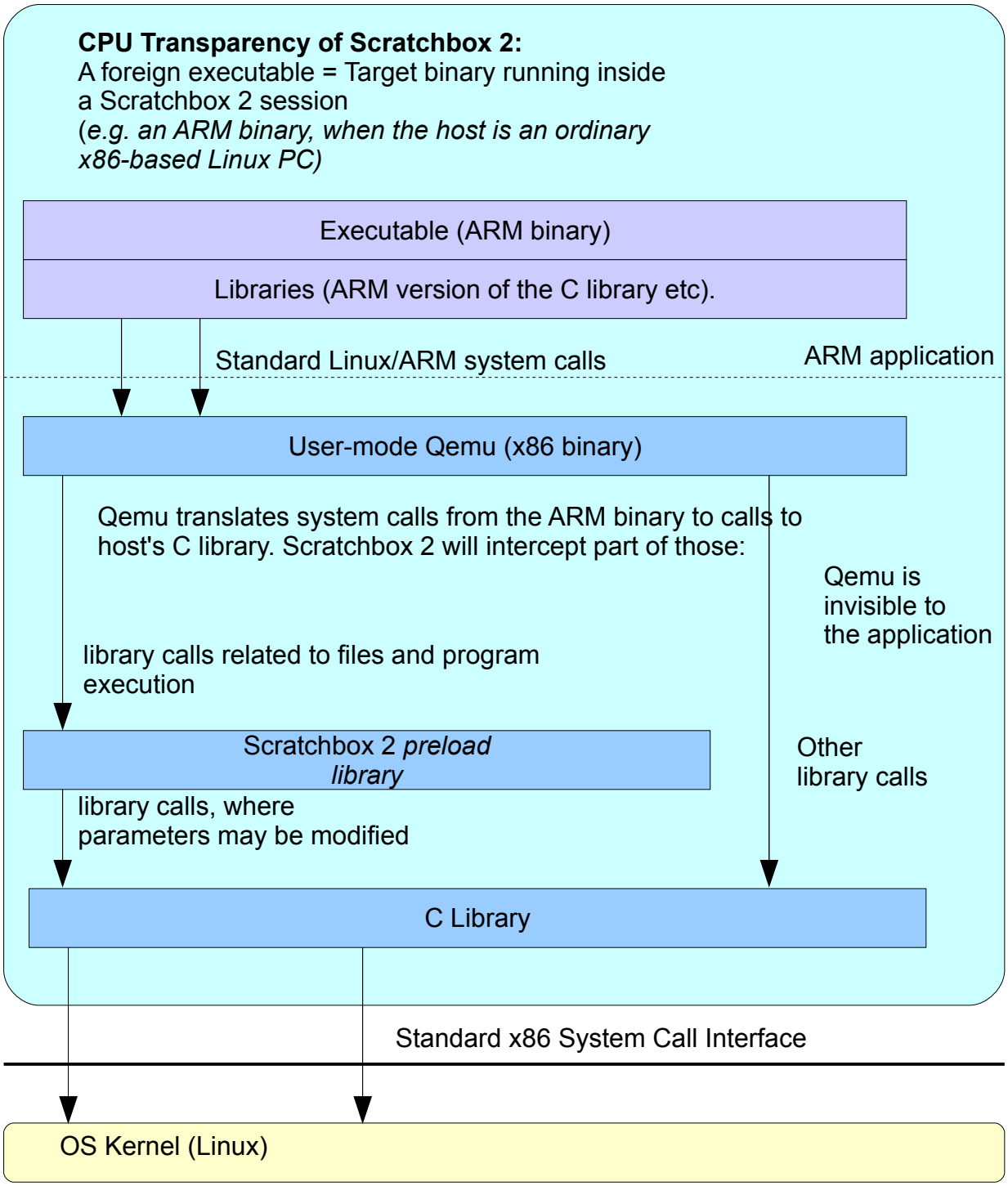

Fig.8. Execution of target device binaries (incompatibe CPU architecture)

3. Interfaces

Most of the time Scratchbox 2 operates in the background. Some SDK implementations may decide to hide it completely from the user; alternatively the user might see that Scratchbox 2 is being used if direct access to the command line interface tools is allowed. From the application point of view, Scratchbox 2 is practically completely hidden: It works by modifying how standard C library calls work. The binary interface only contains calls that are 100% compatible with the existing binary interface standards (e.g. POSIX and glibc ABI), so no Scratchbox 2 specific functions are available.

The next chapters present the basics about the command line interface and the application binary interfaces.

3.1 Command-line interfaces

Command line interface consists of tools for creating and configuring the targets and sessions (*sb2-init, sb2-config* and *sb2),* some diagnostic tools (*sb2-show, sb2-logz*) and wrappers for some software development tools that need to be extended to work effectively in the virtualized environment.

The "*sb2-init*" command is used to create the targets. It must be used once for each target. In the simplest case, *sb2-init* needs just two parameters: Name of the target to be created and path to the cross compiler:

```
$ cd target_root_directory
$ sb2-init ARM_target /opt/tools/bin/arm-linux-gcc
(many messages from the configuration process, and finally:)
sb2-init completed successfully, have fun!
```

Example: using *sb2-init* to create a target called "ARM_target".

Various options are available for fine-tuning operation of *sb2-init.* For example, *-m devel* can be used to set the default mapping mode to *devel* (the default is *simple*). Use `sb2-init --help` to display a full list of available options.

The "*sb2-config*" command is used to select the default target and configure some target-specific settings. See the manual page for details.

The "*sb2*" command creates a session. This is the tool which will be used most, after the target has been configured.

In the simplest form, it needs no parameters: *sb2* will create the session and start a shell, which can be used to execute commands in the virtualized development environment:

```
$ sb2                          <- (This command will create the session)
[SB2 simple ARM_target] lauri@sbox2 workdir $
Notice how the shell prompt has been changed to include the mapping mode (simple), target name
(ARM_target), user and host name and current directory name (workdir).
```

Example: using the *sb2* command to create a session.

Alternatively, if a command is provided as parameter, that command will be executed inside a session. The following example shows how the system's architecture is reported inside the session and how it differs from the real architecture:

```
$ sb2 uname -m              <- (This creates session and executes "uname" in it)
arm
$ uname -m                  <- (This executes "uname" in the host)
x86_64
```

Example: using the *sb2* command to check host architecture.

Operation of *sb2* can be controlled by command-line options. Options *-e* and *-R* are the most important ones: *-e* is a shortcut to the *emulate* mode, and *-R* activates a *fakeroot*-like environment with simulated administrator (root) privileges. *[versions 2.2 actually activate "fakeroot" when -R is used, but 2.3.40 and later ones have a built-in replacement: sb2 got a more comprehensive virtual permission subsystem]*

When used together, these two options also provide read/write access to the target root:

```
$ sb2 -e                    <- (Create a session, using the "emulate" mapping mode)
[SB2 emulate ARM_target] lauri@sbox2 workdir $


$ sb2 -eR                   <- (Create a session, "emulate" mapping mode + simulated root rights )
[SB2 emulate ARM_target] root@sbox2 workdir $
```

Example: Creating a session using the "emulate" mode.

Use `sb2 --help` to display a full list of options.

Other command-line tools include **"*sb2-show*"**, a diagnostics tool (see ch. 4.3), and "***sb2-logz***", a log analyzer and summary generator. Scratchbox 2 includes extensive logging abilities: When the session is created with the *-d* flag (debug), every file access will be logged. Typically, the log grows really fast and can become huge. *sb2-logz* helps by summarizing information from the log.

3.2 Application (binary) interface

For the applications, Scratchbox 2 is completely transparent.

Table 1 presents a list of functions that are intercepted by the Scratchbox 2 core. Most of these are standard POSIX file system operations, in addition the table lists also Glibc extensions that are specific to Linux only (names of such functions usually begin with an underscore).

[version 2.3 intercepts a few dozen more: The virtual permission subsystem affects how getuid(), setgid() etc work, and these are not included in table 1. A comprehensive list can be found from manual page libsb2_interface(7)]

Note that Scratchbox 2 operates on the binary interface level. Applications are not recompiled for it. Part of the Glibc- or Linux-specific functions listed in table 1 are present because the binary interface to Glibc is not necessarily the same as the function which appears in the source-level API. For example, a call to the *stat()* function in the source tends to be changed to a call to *__lxstat()*. The user of Scratchbox 2 does not have to worry about these details, but the developers of it have spent quite many hours figuring out how the Glibc library works internally.

Table 1. API/ABI calls

Public API/ABI calls intercepted by Scratchbox 2 (eglibc			
__fxstatat	dcgettext	getxattr	readlink
__fxstatat64	dcngettext	glob	readlinkat
__getcwd_chk	dgettext	glob64	realpath
__getwd_chk	dlmopen	glob_pattern_p	recvfrom
__lxstat	dlopen	inotify_add_watch	recvmsg
__lxstat64	dngettext	lchmod	remove
__open	dup	lchown	removexattr
__open64	dup2	lckpwdf	rename
__open_2	eaccess	lgetxattr	renameat
__open64_2	euidaccess	link	revoke
__openat_2	execl	linkat	rmdir
__openat64_2	execle	listxattr	scandir
__opendir2	execlp	llistxattr	scandir64
__readlink_chk	execv	lremovexattr	sendto
__readlinkat_chk	execve	lsetxattr	sendmsg
__realpath_chk	execvp	lstat	setattrlist
__recvfrom_chk	exit	lstat64	setrlimit
__xmknod	faccessat	lutimes	setrlimit64
__xmknodat	fchmod	mkdir	setxattr
__xstat	fchmodat	mkdirat	stat
__xstat64	fchownat	mkdtemp	stat64
_exit	fcntl	mkfifo	symlink
_Exit	fcntl64	mkfifoat	symlinkat
_xftw	fopen	mknod	system
_xftw64	fopen64	mknodat	tempnam
access	freopen	mkstemp	textdomain
accept	freopen64	mkstemp64	tmpnam
acct	fstatat	mkstemps	truncate
bind	fstatat64	mkstemps64	truncate64
bindtextdomain	fts_open	mktemp	ulckpwdf
canonicalize_file_name	ftw	nftw	uname
chdir	ftw64	nftw64	unlink
chflags	futimesat	open	unlinkat
chmod	get_current_dir_name	open64	utime
chown	getattrlist	openat	utimensat
close	getcwd	openat64	utimes
connect	getpeername	opendir	wait
creat	getsockname	pathconf	waitpid
creat64	getwd	popen	

4. The Rule Engine: Internal Configuration of Scratchbox 2

The rule engine is a central part of Scratchbox 2. While the user usually doesn't have to write or modify any rules (in fact, a normal user is not expected to even see rules ever), it is still useful to understand the effect of the rules: They define how Scratchbox 2 works, after all.

There are two kinds of rules: Path mapping rules and execution rules. The former are used every time when a file is accessed by name, while the latter are used only when processing the execution-related library calls (*execl(),*etc). All rules are expressed in the *Lua* scripting language. It is a relatively small, embeddable language which provides both a nice way to store the configuration data and to write small extensions to the Scratchbox 2 logic. For more information about Lua, please see *www.lua.org* or the book *Programming in Lua*.

[version 2.3 has a third set of rules, for network address translation]

[version 2.3 contains some important changes to rule engines: Older versions loaded all rules when a process started inside a session, while the 2.3 loads rules only once when session is created and converts them to a binary format, which is then mapped to every process' memory space with mmap(). Some changes had to be done to the rule files (and syntax) because of this, but all basic principles presented in this chapter still apply to the newer SW versions.]

4.1 Path mapping rules

Path mapping rules define how source paths (as specified by the application) are converted to destination paths (real paths that are used by the operating system).

The basic mapping algorithm consists of two steps:

First, the rule needs to be identified. The ruleset contains basically a set of rule lists, that have been organized to a rule tree. This step utilizes fairly standard tree and list lookup algorithms, where the source path is the key. While going thru the rules, the first matching one is selected.

A rule selector specifies what kind of match is required:

- *exact match* requires that the source path is exactly the specified string, for example:
  ```
  path = "/etc/passwd"
  ```
- *prefix match* compares string prefixes: For example, if the requirement is
  ```
  prefix = "/usr/bin/perl"
  ```
  then both `"/usr/bin/perl"` and `"/usr/bin/perl5.8.8"` will match.
- *a directory match* is a special form of the prefix match: It will match only if the specified string is either the source path, or a *full directory prefix* of it. For example, if the requirement is
  ```
  dir = "/usr/lib/perl"
  ```
  then both `"/usr/lib/perl"` and `"/usr/lib/perl/foo"` will match, but `"/usr/lib/perl5"` won't match.

Next, the selected rule needs to be applied. The rule also contains an action, which specifies how to create the destination path from the source path. There are four simple cases, where the destination path is directly deduced from the source path:

- `use_orig_path = true`
  specifies that the path doesn't need to be changed at all; destination path will be the same as the source path, but symbolic links in the path are still resolved (as specified in the path resolution specification)

- `force_orig_path = true`
  is otherwise like the previous action, but it also forces symbolic link resolution to be stopped. There are a few special cases where this is necessary.
- `map_to = "/path"`
  specifies that the destination path will be the same as the source path prefixed with "*/path"*.
- `replace_by = "/path"`
  specifies that the destination path will be created by replacing the matching part, which was used to select the rule with "*/path"*.

Additionally, there are *conditional rules* that require additional conditions to be true before the rule is applied. These are needed to be able to correctly process some special cases, like operation of extendable scripting languages (perl and python) or selecting files based on existence of the file at one of several possible destination directories. These are not presented in detail here; refer to the source code whenever needed.

Consider the following examples:

```
{prefix = "/usr/lib", map_to = target_root}
```

Example: mapping rule 1.

The rule above maps all source paths that begin with */usr/lib* to the target root file system (variable *target_root* contains the location of the target root file system). If *target_root* contains *"/home/user/rootstrap",* then the following mappings will be done due to this rule:

Source path	***Destination path***
`/usr/lib/libxyzzy.a`	`/home/user/rootstrap/usr/lib/libxyzzy.a`
`/usr/lib` (the directory itself)	`/home/user/rootstrap/usr/lib`
`/usr/lib123/libfoo.a`	`/home/user/rootstrap/usr/lib123/libfoo.a`

Example: Results of applying rule 1.

Notice that the since a string prefix match was used to select the rule, it applies to both "*/usr/lib"* and "*/usr/lib123*". Sometimes that might not be what is wanted – it might be better to use a "dir" rule instead:

```
{dir = "/scratchbox/tools/bin",
 replace_by = tools .. "/usr/bin", log_level = "warning"}
```

Example: mapping rule 2.

Rule 2 maps all source paths that refer to directory */scratchbox/tools/bin* to */usr/bin* of the tools collection, replacing the directory prefix of the original path. (Note: the "double dot" operator of Lua is used for concatenating strings)

Assuming that variable *tools* contains *"/opt/tools",* then the following mappings can be done due to the example rule 2:

Source path	***Destination path***
`/scratchbox/tools/bin/foo`	`/opt/tools/usr/bin/foo`
`/scratchbox/tools/bin/subdir/bar`	`/opt/tools/usr/bin/subdir/bar`

Example: Results of applying rule 2.

This rule is an actual example of an Scratchbox 1 compatibility rule. The old Scratchbox 1 used to

keep tools in subdirectories under the */scratchbox* directory hierarchy; it was a convention that is fully specific to Scratchbox 1 and should not be used explicitly. Scratchbox 2 does not use that practice at all. Instead, Scratchbox 2 enforces standard practices whenever possible – for example, tools are located under the */bin* and */usr/bin* directory hierarchies.

However, we have noticed that there are nowadays source packages that contain absolute pathnames referring to files and directories that exists only in Scratchbox 1. Some of these are now supported by compatibility rules, so that the packages can be built also with Scratchbox 2. The rules also contain an additional attribute `log_level = "warning"`, which adds a warning message to the log. The log will be presented to the user when the session is terminated.

As was seen in example 2, rules can also contain attributes that cause side-effects in addition to the mapping. Probably the most important attribute is `readonly`, which causes the destination to appear as write-protected even if the user is the owner of destination and the file system permission flags would allow writing to the file:

```
{prefix = "/lib", map_to = target_root, readonly = true}
```

Example: mapping rule 3.

When a destination has been marked with this attribute, an attempt to open it for writing will cause the call to fail and return error code *EROFS*, which signifies that the file system has been mounted as read only.

The `readonly` attribute is used to enforce file system protection in several rules: For example, it should not be possible to modify the contents of the target root under normal conditions (see fig.4: target root is write protected during the development stage, as well as during program execution – it needs to be writable only during the program installation phase).

Reverse rules were briefly explained earlier in ch.2.3. Internally, Scratchbox 2 uses same kind of rules for forward and reverse mapping of paths. The only difference is that the reverse rules are automatically generated from the forward rules each time when a session is created (forward rules are maintained manually).

4.2 Exec policies and execution rules

The way how programs are started is controlled by *exec policies* in the rulesets. Scratchbox 2 insists that an exec policy must be present when a process wants to execute another program. If the new program belongs to a different "domain", then some settings may need to be changed; the new settings are defined by the exec policy. Also, the currently active exec policy may dictate how some path mappings are performed while the program is running.

In the simplest case, an empty exec policy is enough – the default behavior is enough for many cases. But in a more advanced setup, two or three different policies may be used:

- A “target policy” specifies settings for starting binaries that belong to the target system. For example, if the target system uses the same CPU architecture as the host, the exec policy lists the location of dynamic libraries that should be used (dynamic libraries should not be used from the default locations (*/lib*), because the default directories contain libraries that belong to the host OS)
- A tools policy specifies settings for “tools” (see fig.5).
- A host policy may also be used, it practically just says that no special settings are needed for programs that belong to the host OS.

The exec policy may be selected by the mapping rule which is used to map the pathname of the

program to be executed. However, to avoid adding the `exec_policy` attribute to every mapping rule, a policy selection table can be used. It specifies exec policies based on locations in the file system (based on real locations = destination paths, not on source paths).

Following examples are from the "devel" mapping mode. Variables at the right side of each assignment depend on the target system, and values are assigned to them whenever a session is created:

```
exec_policy_target = {
   name = "Rootstrap",
   native_app_ld_so = devel_mode_target_ld_so,
   native_app_ld_so_supports_argv0 = conf_target_ld_so_supports_argv0,
   native_app_ld_library_path = devel_mode_target_ld_library_path,
   native_app_locale_path = conf_target_locale_path,
}
```

Example: exec policy 1.

A rule which closely resembles mapping rules is used to glue this exec policy to all binaries that need it:

```
{ prefix = target_root, exec_policy = exec_policy_target }
```

Example: Binding rule: From destination path to exec policy.

As can be seen, this policy defines only settings for "native applications", i.e. applications that have the same CPU architecture as the host computer. Incompatible target architectures are actually slightly easier to handle: Less configurable items are needed, the target can be identified by reading CPU type from the executable, and the CPU emulator can usually set up things quite well without lots of external configuration items.

The exec policy may also be needed while the program is running: There are cases, where pathnames need to be mapped differently, based on the type of the binary which is running. (this concerns the development modes, i.e. the "blue" side of fig.4). The two most important cases are

1. Handling of the "locale" database, including the message catalogs. In practice this means that mapping of */usr/share/locale* depends on the type of the running program. Target and tool binaries will need to be map it to their respective locations.
2. Handling of script language extensions, namely, libraries and extensions of *perl* and *python.* For example, */usr/lib/perl* must be mapped to the target root when target's perl is running, but to the corresponding place in the tools directory when the other perl is running.

4.3 Testing out and displaying the results of mapping

Scratchbox 2 has an utility program which can be used to examine mapping results: *"sb2-show"* performs the mappings, and then displays the results without actually using them. Note that *sb2-show* can only be used inside a session:

```
$ sb2                          <- (create the session)
[SB2 devel ARM_target] lauri@sbox2 workdir $ sb2-show path /usr/include
/usr/include => /home/lauri/rootstraps/ARM/usr/include (readonly)
```

Example: using *sb2-show* to display pathname mappings.

Also, *sb2-show* can be used to examine what will be actually executed when one of the exec-class funtions is called:

```
[SB2 devel ARM_target] lauri@sbox2 workdir $ sb2-show exec /usr/bin/make
(this produces 17 lines of output when executed on the author's development machine – the relevant part is
that "make" is executed as host-compatible program)
[SB2 devel ARM_target] lauri@sbox2 workdir $ sb2-show exec ./testprogram
(shows how "./testprogram" will be executed  with Qemu)
```

Example: using *sb2-show* to display how programs get executed, really.

5. Scratchbox 2 “core”: The Preload Library

Internal operation of Scratchbox 2 is based on preloading a library to each running process. That library, called *libsb2.so* or simply the “core”, contains everything that is needed at runtime. All mapping operations etc. are performed by the core library: Scratchbox 2 does not use daemons or other external components that would be contacted on demand.

[version 2.3 added a daemon (“sb2d”) but it isn't used for mapping operations: Instead it is used for updating the virtual permission database, which replaced “fakeroot”. And the amount of IPC with the daemon was minimized: Processes are able to read virtualized permission information without contacting the daemon]

The mechanism which is used to load such libraries is called *library preloading*. It is a standard feature of most modern Unix-like operating systems[11]. In practice it means that the dynamic linker/loader (*ld.so*) which loads programs to memory will also load the preload library and use symbols of that library to override symbols from the default libraries; Scratchbox 2 uses this possibility extensively.

Architecture of the core library is presented in fig.9. Following chapters will briefly present the purpose and function of each component of the *libsb2.so* library.

[other changes in version 2.3: The Lua interpreter has been eliminated from libsb2.so => all modules which are marked as “(Lua)” in fig 9. have been re-implemented in C. Also, there are new subsystems: Virtual persmission system and network address translation are not present in fig.9, which refers only to Scratchbox 2 versions 2.0 – 2.2]

11 At least Linux, Solaris and Mac OS X are known to support this mechanism. However, currently Scratchbox 2 can only be used on Linux – porting it to other platforms should be quite easy. But remember that other components are also needed to create a fully functional SDK – currently the lack of decent target emulator for other operating systems seems to be the biggest obstacle to the porting work (the user mode of Qemu is still quite Linux specific – it does not emulate system calls, it just translates them from the target architecture to calls to host system kernel)

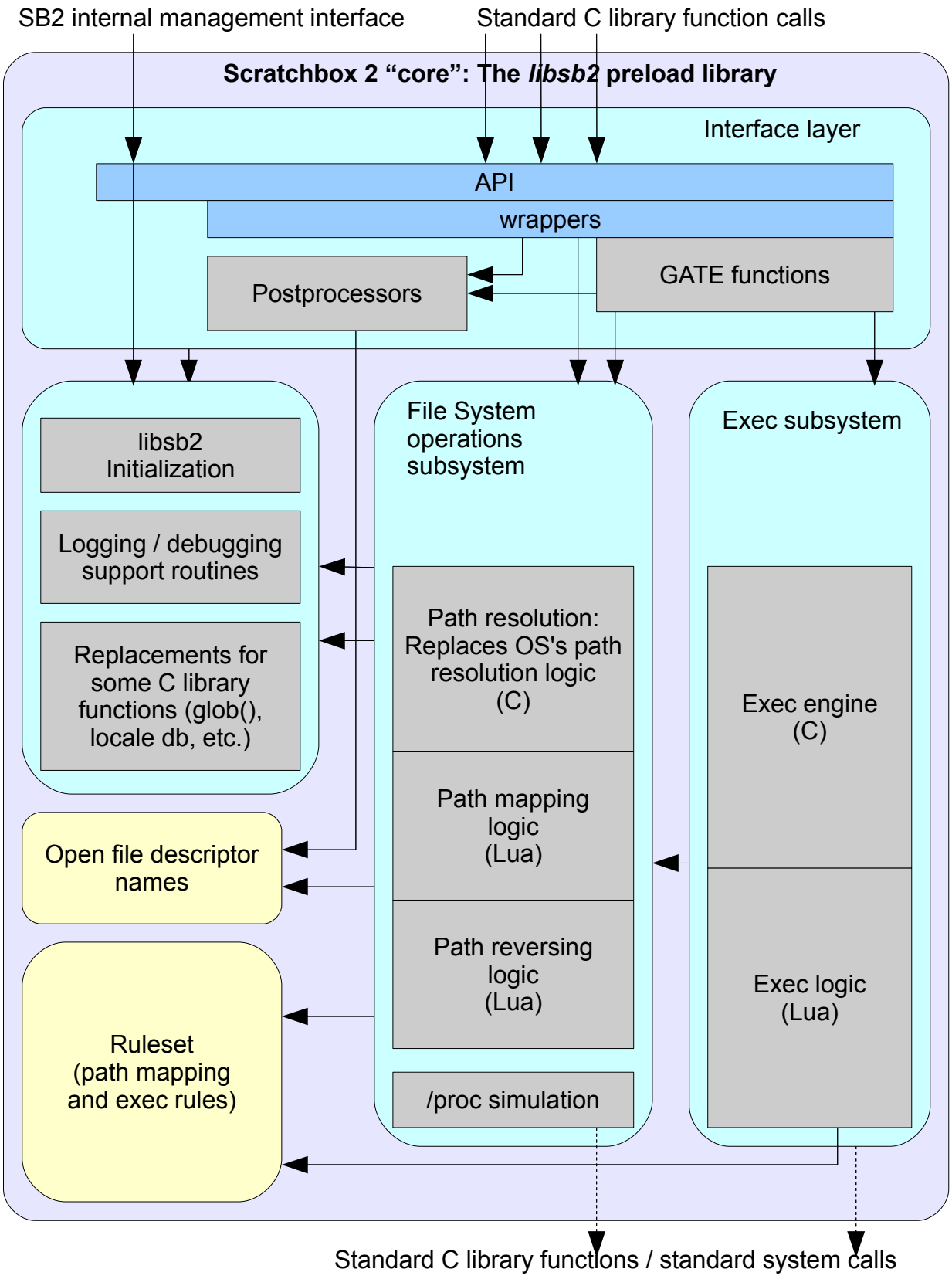

Fig. 9. Architecture of the Scratchbox 2 preload library = "the core"

5.1 The interface layer

The interface to the Scratchbox 2 core library is completely defined by an interface specification file. Both the actual interface code and the list of functions that will be exported from the library are generated with a small interface generator tool (included with the Scratchbox 2 source code).

Three types of definitions can exist in the interface specification file: API/ABI calls from table 1 are defined as needing either a “wrapper” or “gate” function. Wrappers are used when the interface generator is capable of creating everything that is needed to handle a function, and gates are used for more complex cases. Additionally, internal management interface functions (used by *sb2-show*) is defined as “exported” functions in the specification file.

A wrapper performs specified parameter transformations (usually path remapping) and then calls the “real" function with the same name from the C library (glibc). For example, this is all that is needed to get the *stat()* function handled by Scratchbox 2:

```
WRAP: int stat(const char *file_name, struct stat *buf) : map(file_name)
```

Example: Interface specification for the *stat()* function

The example above defines that the wrapper utilizes the *map()* modifier; parameter *file_name* points to a pathname which is subject to path mapping. The interface generator creates C code which calls the mapping logic for the parameter, calls the logger if logging is enabled, looks up the real function from the C library, calls it, and finally returns result to the application. Nothing else is needed in this case. Majority of the functions listed in table 1 can be handled with wrappers.

“Gates” are used for the remaining, more complex API functions: These are otherwise like wrappers, but the real function from the C library is not directly called. Instead the generated code makes a call to a hand-coded function, which may do additional steps before the real function is called or even replace the functionality of the real function completely. For example, this definition creates a gate function for *execl():*

```
GATE: int execl(const char *path, const char *arg, ...) : pass_va_list
```

Example: Interface specification for the *execl()* function

The automatically generated part will do similar preparatory steps as a wrapper will do, but once all such preparations have been done, the generated function will make a call to *execl_gate(),* a hand-coded function which belongs to the exec subsystem. *execl_gate()* may then decide if it calls the real *execl()* or does something else instead (in fact, it does “something” else – see ch.5.3. below).

Another modifier can be seen in the second example above: *pass_va_list* marks that a variable argument list is present and needs to be relayed as a *va_list* object. The interface generator accepts about a dozen other modifiers for wrappers and gates. Please see the source code of the interface generator (*gen-interface.pl*) for details. All possible modifiers are documented in a long comment in the beginning of that Perl script.

5.2 File system operations subsystem

File system operations subsystem is responsible for all path mapping operations. It consists of two layers:

- First one is a replacement for the *path resolution*[12] logic which usually belongs to the operating system. Scratchbox 2 must provide a replacement for it to make symbolic links work correctly. This part has been implemented in C, as there was little implementation

12 See chapter 2.3 and manual page *path_resolution* of Linux

freedom – it must closely mimic operating system's behavior.

- Second layer is the actual path mapping engine, which executes the mapping rules. In versions 2.0 - 2.2 it has been implemented in Lua *[version 2.3 implements it in C]*. Path mapping engine has two sides, the forward mapping logic and the reversing logic.

The architecture diagram also shows a module called */proc simulation*. Purpose of it is to provide replacements for some files from the */proc* special filesystem that would otherwise be incompatible with the rest of the virtualized environment:

For example, */proc/self/exe* is a special symbolic link which points to the location of the running program in the filesystem. Without the */proc simulation* module programs could find out from it the real locations of executables. And, typically some Perl scripts do read the value to find out location of the Perl interpreter. Even if the real */proc/self/exe* might point to something like */home/user/rootstrap/usr/bin/perl*, the running binary must see it pointing to */usr/bin/perl* and that is just what the */proc simulation* module does: It fixes the problem by providing a replacement symbolic link which points to the reverse-mapped Perl interpreter location.

5.3 Exec Subsystem

The execution subsystem handles the "exec family" library calls: *execl(), execle(), execlp(), execv(), execve() or execvp()*. Processing of these functions is not as straightforward as plain path mapping is, because both the type of the binary to be executed and the exec policies can affect the results. Just as was the case with the path mapping subsystem, the exec subsystem is also divided to two layers: "engineroom" which is implemented in C, and also here the actual exec mapping logic is written in Lua *[version 2.3 implements it in C]*.

Brief description of the algorithm follows, with built-in references to the implementation:

First, *Exec preprocessing* takes place. The purpose is to determine *what file* needs to be executed. And that might very well be something else than what the application requested. Arguments may also be added, deleted, or modified during preprocessing. For example, an attempt to run */usr/bin/gcc* will be replaced by a path to the cross compiler in the development-oriented mapping modes (e.g. the result may be */opt/tools/bin/arm-linux-gcc*). The cross-compiler also typically needs a few more flags to work correctly (please use *sb2-show exec* to see the actual outcome)

Second, the system needs to determine *where the file is,* i.e. the real destination path is calculated by making a call to the regular path mapping engine. A notable side-effect of that is that the mapping logic may also return name of the exec policy which will be used during step 4.

Third, once the real location of the file is known, the system finds out the type of the program. This involves opening the file and examining it.

Step four consists of deciding *how to do the actual execution*. There are basically three possible cases:

1. For scripts (files starting with #!), script interpreter name will be read from the file, mapped to get the real location of the interpreter, and then the exec processing logic is restarted from step 1 for the interpreter, with the path to the script as an argument.
2. For native binaries, an additional call to an exec post-processing function will be made, because the type of the file is not enough to specify the environment that needs to be used for the file: This is the place where the exec policy rules will be applied. There are at least four different cases where additional settings may need to be applied:
   2a) Native binaries that are compiled for the host system can be started directly
   2b) Programs from the tools collection may need to load dynamic libraries from a different

place (e.g. */opt/tools_root/bin/ls* may need to use libraries from */opt/tools_root/lib*, instead of using them from */lib*). This is implemented by making an explicit call to the dynamic loader (ld.so) to start the program, with additional options for *ld.so*.
2c) If the target architecture is the same as the host architecture, binaries may also need special settings. This also uses an explicit call to *ld.so*.
2d) Static binaries may be executed directly, or by another Qemu (depends on configuration)

3. Target binaries (when target CPU architecture is incompatible with host CPU architecture) are started in "cpu transparency mode", which typically means that Qemu is used to execute them.

Step five is to make a call to the real exec function, *execve()*[13], which will make the system call to the kernel.

5.4 Internal databases: The ruleset and the open file descriptor table

The Scratchbox 2 core needs two internal databases: The first holds the ruleset, which is practically constant information, and the second one is *open file descriptor table*, which is a lookup table from file descriptor numbers to pathnames. The latter is needed for proper implementation of **at*-class functions (*openat(), fchmodat(),* etc). Performing correct pathname mappings for these functions is only possible when the file descriptor number can be converted back to a file name.

[version 2.3 keeps the ruleset, as well as several configuration variables in a simple shared memory database, the "ruletree". It can be printed out with utils/sb2-ruletree *for debugging purposes.]*

5.5 Supporting routines: Logging, initialization etc.

The Scratchbox 2 core library is automatically initialized when it is used. No explicit initialization is required.[14]

Scratchbox 2 core also provides quite extensive logging capabilities. The logger is normally always active, but only messages at logging levels *error* and *warning* will be written to the log. For diagnostics purposes logging level can be changed at session creation time; check the *-L* and *-d* options of the *sb2* command line tools.

13 That is right: All six different exec family functions end up calling *execve()*. The other five are practically only shortcuts to *execve()*.

14 In fact, it was found out the hard way that even if the dynamic library system makes it possible to introduce library constructor and destructor functions (called *_init()* and *_fini()*), these are not useful in practice: Execution order is not guaranteed and the library may be entered even before the constructor function is called. Fortunately it was easy to add protection against this situation, thanks to the completely generated interface layer.

6. Native Binaries, LD_PRELOAD and Scratchbox2

While execution of *foreign* binaries with Qemu in Scratchbox 2 is straightforward, easy, and relatively safe, *architecturally compatible binaries* have presented some more or less hard challenges. Solutions to these should be understood by anyone who wants to implement a full-featured SDK, or whose target system is architecturally compatible with the host. Even if the target uses a foreign architecture, the tools that are used with Scratchbox 2 might not be compiled against the host system libraries and are treated as *target-style architecturally compatible binaries*.

The term *architecturally compatible binary* refers to a program that is directly compatible with the host system CPU: A binary that can be executed without any CPU emulation, or even outside of Scratchbox 2 sessions in some cases[15].

Linux, and eglibc/glibc (standard C library) in particular, present some challenges to proper execution of such programs. This chapter contains brief descriptions of these challenges and also introduces the solutions that are currently in use within Scratchbox 2. Some solutions are based on small patches to *eglibc/glibc*; these patches are shipped with Scratchbox 2 sources (in the *external-patches* directory). The patches are generic enough so that they could be added to the official sources, but that hasn't been done yet.

6.1 Static binaries

Scratchbox 2 depends on the dynamic library preloading mechanism. Naturally, that is not available with static binaries - which don't load any dynamic libraries anyway. Fortunately this is a minor problem in practice: Almost all Linux programs are dynamically linked.

There are three well-known exceptions:

- The dynamic linker itself is a statically linked program. Environment variables and command line options can be used to alter some operations, but unfortunately not all: The standard dynamic linker does not have enough features for precise operation that Scratchbox 2 needs, but adding proper support to it was not a difficult task to do. The next section describes this in more detailed level.
- *ldconfig* is a statically linked binary (it is used to maintain dynamic library configuration in the system). By default, it uses absolute pathnames to access the configuration files.
  A really easy workaround exists: There is a command-line option for *ldconfig* that can be used to specify a prefix which will be added to the configuration file pathnames. Scratchbox 2 provides a wrapper which will add that option whenever *ldconfig* is executed.
  The tough part is that latest versions of *glibc* have been shipped with a buggy *ldconfig*: Just this option has been broken so that it only works for the *root* user. A patch for this is shipped with Scratchbox 2 sources; it is needed for *glibc* 2.8 and later, at least (the option works in that ldconfig which shipped with glibc 2.5; we don't know about versions 2.6 and 2.7).
- *Valgrind* is a software analysis tool that is statically linked. It is currently incompatible with Scratchbox 2; *chroot* must still be used with it.

15 If the host system contains all dynamic libraries that are needed, it is possible to just call one of the *exec* functions and the binary will be started outside of Scratchbox 2. However, the host system provides the dynamic linker in this case, and that will use the dynamic libraries from the host. Those are not necessarily in sync with the target system. We have learned that in most cases, but not always, the programs seem to run without problems...but sometimes they do something unexpected (crash, maybe).

Why this is not a problem with foreign binaries: Foreign binaries are executed with Qemu, which is a dynamically linked binary itself, and relies on the Scratchbox 2 core. Qemu translates all system calls that the foreign binary makes – including calls from statically linked foreign binaries – and ends up calling the relevant functions from the Scratchbox 2 core in any case.

6.2 Challenges caused by the dynamic linker (ld.so)

There are two ways how the dynamic linker can be started:

First, it can be started implicitly by the Linux kernel. Each dynamically linked program contains path to the dynamic linker that should be used (for example, for the x86 architecture, that path is usually */lib/ld-linux.so.2*). The kernel finds that path from the file, maps the dynamic linker to memory, and jumps to it.

The second way of starting the dynamic linker is to start it explicitly, just as any other statically linked program is started. In this case the path to the program that will be loaded is provided by a command line argument. This method was probably originally intended for testing purposes, but it is in fact the only way how execution of architecturally compatible binaries can be done safely: Such binaries might contain the exactly same, absolute path to the dynamic linker as what is used on the host system. However, typically host's dynamic linker can not be used to load target's binaries when target's libraries must be used[16]

So, when the exec policy says that a host-compatible binary really belongs to the host, Scratchbox 2 uses the first approach and lets the kernel start the dynamic linker. But when the policy specifies that the binary belongs to the target, the second way is used. Target's dynamic linker is started explicitly. This presents some additional challenges:

First, the dynamic linker wants to access some configuration files while it is loading the libraries. The stock version didn't have any options to change all of these paths; this is one thing that had to be patched.

Second, the files that are loaded (both the binary and additional libraries from the target system) may contain even more absolute paths: “RUNPATH” and “RPATH” options inside the files can point to directories, which will be searched during the library lookup process, in addition to what the LD_LIBRARY_PATH environment variable specifies. There was no way to alter these paths, but we have added an option for it as well.[17]

The third challenge was related to semantics of the *exec* calls. The problem is that the explicit, command-line way of starting the dynamic linker looses one parameter that was intended for the application. There is a patch for this problem, too (this is the so-called *argv0 option patch).*

What happens without our patches: Scratchbox 2 detects (at session set-up time) if the target system has a dynamic linker that has the patches. If it doesn't, the linker is still used, but some things may not work as expected (e.g. files that contain RPATH options may cause libraries to be loaded from the host filesystem, etc)

16 glibc's dynamic linker seems to be very incompatible with other versions of itself – that is, if your glibc's version is x.y.z, you *must* use dynamic linker which was shipped with that version. Host's linker can be used only if the C library comes from the host, too. This is yet another thing that was found out the hard way during development.

17 The stock ld.so does have an option (*--inhibit-rpath*) for ignoring RUNPATHs and RPATHs. It is certainly better than nothing, but not enough, and can be easily applied only to the binary itself – not to libraries that are loaded indirectly. Anyway, Scratchbox 2 uses that option if the detects that the dynamic linker does not have our *–rpath-prefix* option.

6.3 Built-in absolute pathnames in glibc

Glibc contains many functions that will open files or do something else with built-in absolute pathnames. However, the core library of Scratchbox 2 can only make operations with pathnames that are generated outside of glibc – it can not alter anything that is private and internal to Glibc itself (see fig.7). It can be said that the Scratchbox 2 core is at too high level in the software stack – the correct place would be between the system call interface and the C library, but that can't be done with the current preloading mechanisms and current library structure (because the system calls are implemented inside glibc)

One example about this problem is the “locale” database: Glibc loads error messages etc. language-specific strings from the “locale” database. The default location of the database has been built into the library. This is not acceptable, because the target- and host systems might need different databases.

Fortunately the “locale” database can be easily relocated without any changes to the library. An environment variable exists for this: It can be used to define the path to the database[18]. Scratchbox 2 sets that variable automatically always. There are some other subsystems, that are treated like that, too.

However, not all subsystems of glibc have this property. One example is that path to the name system service configuration file[19] is hard-coded to Glibc, and couldn't be changed by any environment variables. We have introduced a new variable for this (yet again, the patch is shipped with Scratchbox 2 sources).

Why this is not a problem with foreign binaries: Note that even if foreign binaries also use Glibc, which has been compiled from the same sources and suffers from the same anomalies, there are no problems with it. In this case the Scratchbox 2 core operates at lower level; all pathnames that are passed from the C library to Qemu, and path translation takes place after that (see fig.8)

6.4 Versions of glibc and target-specific Scratchbox 2 core libraries

The host operating system and an architecturally compatible target system might use different versions of the C library; this is quite natural.

The way how C programs are compiled, how the glibc has been designed, and how the Scratchbox 2 core library needs to interact with the C library, all mean that the Scratchbox 2 core is somewhat tightly bound to the C library. This has presented some binary-compatibility challenges (there are questions about library versioning, etc), but there is also a really easy solution that has practically eliminated all binary-compatibility issues:

The best way to ensure full compatibility between the Scratchbox 2 core and the target system's C library is to use target-specific versions of the Scratchbox 2 core library for each target. This means that the core library should be compiled separately for the host and the target systems, starting from the same sources. Scratchbox 2 will detect such target-specific core libraries at session setup time, and can execute different programs with different sets of libraries (based on the exec policy settings. Details about how the setup is done can be found in the “sb2” script.)

18 Unfortunately there are two formats for the “locale” database: packed and unpacked. The environment variable can only point to an unpacked database; There is no obvious reason why Glibc has this limitation, but it means that packed databases must be expanded before the “locale” databases can be used with Scratchbox2.

19 Usually */etc/nsswitch.conf*.