University of Hamburg
Department: Computer Science

Research Paper

Generate a Reproducible Environment for Spack

By: Tim Rolff

Supervised by: Dr. Michael Kuhn

Contents

1 Introduction

Managing or compiling software in an High-Performance-Computing (HPC) environment is often a challenge. Therefore it is useful to provide software which manages and abstracts these repetitive tasks. A great tool, which is especially designed for the HPC environment, is Spack [Gam17b], [Gam16]. While Spack aims for simple usage and usability it has some flaws when it comes to the management of jailed build environments. In this paper I want to address this issue and propose / implement multiple solutions to solve this problem.

2 What is Spack?

Before I go into details, I want to briefly introduce Spack [Gam15]. Spack is designed as a package manager for High-Performance-Computing environments such as the Deutsches Klima Rechenzentrum (DKRZ) or the Livermore Computing Center [Gam16]. While most HPC-Centers have their own tools and software to manage their environment, many additional software depends on libraries and specific compilers. Further to the dependencies of the used software each user has different needs as well, as for example different libraries or program versions and build systems. With these, there are also often specific compilers for MPI or CUDA required, which also depend on the users needs. Managing all these is quite a challenge, which often results in a time consuming tasks to compile the newest software. This is where Spack comes into place, Spack tries to abstract all libraries into simple packages which contains the information about its build system and the dependencies to build a specific package. Furthermore Spack manage the available compilers, which can be used to compile the specific package. While Spack manage the underlining build systems, it does not try to replace them (see figure 1). This ensures that the user can use any build system, if it can be accessed by command line or an API. The build systems, which are already supported by Spack are Autotools[1], CMake[2] and Make[3], as well as some other build systems for R and Python (see [Gam17d]). Further Spack offers the functionality to support multiple packages of the same library with different compile options and / or versions and / or compilers. While this can be also used to reduce compiletime by reusing the already compiled packages. The main benefit is to avoid a collision with a wrong linked library or program. This

[1]`https://www.gnu.org/software/automake`
[2]`https://cmake.org`
[3]`https://www.gnu.org/software/make/`

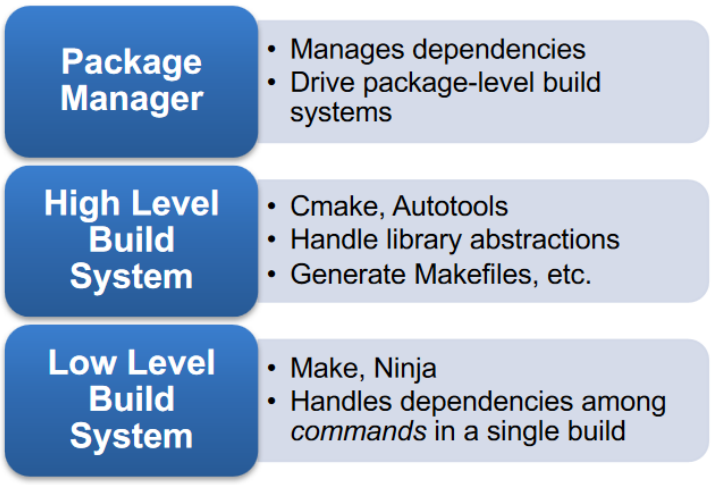
Package Manager
• Manages dependencies
• Drive package-level build systems
High Level Build System
• Cmake, Autotools
• Handle library abstractions
• Generate Makefiles, etc.
Low Level Build System
• Make, Ninja
• Handles dependencies among *commands* in a single build

Figure 1: The Spack hierarchy. On top is the Spack implementation, which manages dependencies and build systems. Underneath are the typical build systems. Source: [Gam16].

also results in reproducible builds, which might be important for some users, especially in the case of scientific research.

While most package systems are already build into the operating system (e.g. apt, pacman, yum for Linux distributions) or get shipped with other programs (e.g. pip for python) Spack tries a different approach by being as lightweight as possible. This allows the user to use Spack in the home directory without any special privileges. which is often the case, when installing software with a regular package manager. Therefore it is only necessary to clone Spack from its repository into the home directory via:

```
$ git clone https://github.com/LLNL/spack.git
```

This downloads all necessary files to run Spack itself, without installing it into the typical Python environment by running setup.py. Although Spack itself is independent from the Python environment, it is necessary to have Python along with some tools such as a compiler, tar, bz, unzip etc. and libc installed in order to run Spack and build packages with it.

2.1 Basic Usage

To install a package it is only necessary to run [Gam17b]:

```
$ ./spack install bash
```

This example compiles and installs the latest version of Bash, which is known to Spack. Further the command ensures that the necessary dependencies ncurses and readline get compiled before the compilation of Bash. It also guarantees that these libraries get linked into the binary via an rpath or a static library.

It may not be always sufficient to use the latest Bash version, therefore it is also possible to run

```
$ ./spack install bash@4.3
```

to install Bash as version 4.3. As mentioned before, Spack makes sure, that both installed Bash versions can coexist. To ensure that there is no collision between two packages of the same name, Spack assigns a hash to each configuration. It then get their own installation directory. In addition to these simple commands, there is the possibility to set the custom compiler by

```
$ ./spack install bash@4.3 %gcc@7.2
```

or by setting additional compiler flags through

```
$ ./spack install bash@4.3 cppflags=\"-O3\"
```

For other possibilities see [Gam17b] and especially [Gam17c].

2.2 Package Creation

Such as the most package systems, Spack offers the possibility to create custom packages [Gam17b]. These could be configured, to fit the needs of individual HPC environments. An example for such a package can be seen in listing 1.

```
class Bash(AutotoolsPackage):
    url      = "https://ftp.gnu.org/gnu/bash/bash-4.4.tar.gz"

    version('4.4', '148888a7c95ac23705559b6f477dfe25')
    version('4.3', '81348932d5da294953e15d4814c74dd1')

    depends_on('ncurses')
    depends_on('readline@5.0:')

    def configure_args(self):
        spec = self.spec
        return [
            'LIBS=-lncursesw', '--with-curses',
            '--enable-readline',
            '--with-installed readline={0}'
            .format(spec['readline'].prefix),
        ]
```

Listing 1: Example for a simple package
which is shipped with the Spack package manager.
Source: `https://github.com/LLNL/spack/blob/develop/var/spack/repos/builtin/packages/bash/package.py`, visited on 19.09.2017

A package file currently consist of a directory inside the Spack dictionary structure and a Python file, which contains all necessary information to build the package. For this given example, the basic package type is already defined by the base class. Hence the example package is an Autotools package, which already calls most of the required steps for the build process. In order to download the given package the url is specified by the url parameter of the class and the given version number. In most cases Spack tries to extrapolate the new URL from the given one, by replacing the version number. But it is also possible to specify the URL for each version independently. In addition to the version number, the package also defines its dependencies by the depends_on command which are used to create the dependency graph. In this example the package requires some specific compile time arguments. They can be specified within the configure_args function. This sets all the necessary packages for the Bash build system, such as the installation directory of the previous compiled readline.

In order to create a package with the necessary information to compile the program, it is necessary to call

```
$ ./spack create url
```

which creates the necessary boilerplate code to write a custom package (see [Gam17b] and especially [Gam17a] for more details).

3 Current Problems When Building Packages

Spack usually works without any difficulties when compiling a new program, under the condition, that it does not depend on a already installed library. It has some problems when a library is already installed with Spack and also installed on the system with the distribution package manager. If the package can not be found by the compiler at the specified path, inside the Spack directory, the compiler searches at other locations to find a similar library with the same name. This may happen if the library file got deleted, without deleting the directory, which contains the library, or by not using Spack uninstall. This then may result in a wrong linked library. This behavior leads to an runtime error, if the library can not be found on the device which runs the program. Even if the library can be found, the results of the program may be wrong, because it might not result in the expected behavior. Hence the expected result of a function from the library might be different, or does not contain the necessary bug fix. At the least worst case, the program does not compile at all or the used library is exactly the same. All the mentioned cases also happen if the creator of a package forgot to define the correct dependencies (see previous chapter). This may also result in the same problem, that the package might compile only on a few systems, which happened to have the specified dependency installed and fail on other systems.

This is not a directly fault of Spack, but rather a general usability feature of the compiler such as the GCC. The feature allows the specification of a library by its name without explicitly specify the location. In case of the GCC, this command is specified through:

```
$ gcc -L/path/to/libraries/ ...
```

While this does not search through all files on the system, there are some general locations built in the compiler, such as for the GCC [JM05]:

```
/usr/local/include
/usr/include
/usr/local/lib
/usr/lib
```

These may vary for different compiler versions and different operating systems. However the compiler is allowed to use the libraries in those paths. Even if the compiler was compiled without any default paths, the user may set custom paths with the environment variable **LIBRARY_PATH**[4]. Therefore it is necessary to omit the default compiler search paths, to search in locations which are not part of the default operating system libraries.

A way to achieve this, could be the modification of the compiler. But because there exist huge variety of different compilers for different architectures and systems which are supported by Spack, it would require to change all of them. In some cases it might not even be possible to rewrite parts of the compiler. As for example for the Intel C++ Compiler, which is a closed source commercial product. Therefore changing the compiler is not a feasible solution to the problem.

Another way is jailing the compiler inside an deterministic and well known environment such as a plain Linux distribution or a custom one. This has the benefit, that the compiler always work in an minimal environment, without knowledge about existing system or third party libraries. Additionally this introduces another security layer to compile and run untrusted source code.

A lighter approach would be the partial visibility of only the necessary files to the compiler. This would use the operating system of the host system which should already provide all necessary system libraries. While the opposite solution would be the usage of a virtual machine (VM) which simulates a complete system with an environment. This would also enable using different processor architectures.

[4]`https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Environment-Variables.html`

4 Implementations

In the following sections I want to address some possible solutions to the problem, by presenting the implementation and the idea behind it, as well as examine the pros and cons of each approach.

4.1 Basics

4.1.1 Chroot

The idea behind the chroot command is to change the root directory of a process to a new path. This ensures that the specified path contains all child nodes and additionally it makes sure that it is the highest path node in the file tree (see [Ker17a] and especially [Ker17b]). This allows for jailing the compiler inside a generated chroot environment to restrict it to files, which are also inside the jail. A positive aspect of the chroot command is its existence since glibc 2.2.2 (see [Ker17b] and `https://ftp.gnu.org/gnu/libc/`). This implies, that it should be fairly common inside a modern Linux environment. In contrast to the common availability of chroot, it introduces a huge restriction. The usage of chroot is restricted only to users which have the CAP_SYS_CHROOT capability. This capability requires administrator rights on some systems, as for example on Debian systems[5]. To workaround this restriction it is possible to unshare (see [Ker17a] and especially [Ker17j]) the process before calling chroot, but this is only possible on Linux kernels with version 2.6.16 or newer.

While the chroot command is not considered to be unbreakable, because it is possible to leave the environment by moving files outside, it is secure if all files stay inside the jail [Ker17b]. This allows to compile untrusted source code or programs inside a static jail which does not require moving files in- or outside.

On a more recent system, which builds upon the new systemd init daemon, it might be reasonable to use the newly introduced systemd-nspawn (see [Ker17a] and especially [Ker17i]). This command supports more advanced features, such as the support for running a different operating system inside a container. It also offers the possibility to select between different file types, such as an system images or a dictionary with all system files. Additionally it can initiate the init binary as an lightweight alternative for a virtual machine.

[5]`https://wiki.debian.org/chroot`

4.1.2 Mount Bind

As mentioned before, when using chroot it is usually not possible to access files outside the jail. While this is the expected behavior, it introduces some problems, when dealing with devices files, which should be defined inside the /dev directory. These files are required for special functions such as random and urandom (see [Ker17a] and especially [Ker17h]) which are required for multiple applications or other system services. Therefore in order to create a fully usable environment it is necessary to access these devices. Beside the need for devices, it is also necessary to allow the jail the connection to the network. This allows Spack to download the required packages which are defined in the URL parameter (see chapter 2.2). Additionally some applications, which are part of Spack need access to these in order to work correctly. As a solution to this problem it is possible to remount parts of the file tree into another child node (see [Ker17a] and especially [Ker17d]). This can be realized via the

```
$ mount --bind
```

command. It allows the bound directory to be visible inside the chroot jail and enables the support to read and write into the files, which are contained inside the directory. As with chroot the mount command requires a capability. In the case of mount this is even more restrictive, because it requires the CAP_SYS_ADMIN capability. This capability, as the name implies, is directly defined as administrator rights and therefore results in a huge restriction for non administrator users.

4.2 Implementation Overview

4.2.1 Spack Structure

To extend Spack it is necessary to understand its structure. Fortunately Spack has a very clean design, which aims for easy extensibility. In a coarse view Spack consists out of two main parts, the implementation and a list of packages. To extend Spack, it is only necessary to extend the implementation part. Consequently I will only focus on the implementation which is inside the lib/spack/spack path of the Spack directory structure. This directory contains the list of all available commands (inside the cmd/ directory) as well as the entry point for the main Spack implementation (__init__.py).

4.2.2 Utility Functions

Because Spack already handles the most implementation details, it is only necessary to create a utility script which provide all necessary functions. With these functions it is then possible to jail Spack and to write user interface commands. Ideally the utility script should provides the commands to construct or destroy an environment. Hence it would be useful to provide the following functions:

```
def build_chroot_environment(dir)
```

This should handle the generation of the jail or VM for a specific directory. It should be called before isolate the environment. Further it should handle the calls to mount bind or copy all needed files in order to generate the environment.

```
def remove_chroot_environment(dir)
```

Command to destroy the environment without removing it. This should unmount all mounted files in order to avoid data loss in the case that the environment should be totally removed.

```
def isolate_environment()
```

The isolate command should create the chroot jail and handle the calls to the virtual machine / container system. Furthermore it should call the Spack implementation inside the constructed jail. It should also execute the commands which are specified in the command line with the jailed Spack implementation. After the construction of the environment, it should be only necessary to call this function in order to create the jail. If not all requirements are fulfilled, the function should handle the calls to generate the environment by calling build_chroot_environment.

While these functions should be implemented in all featured implementations, they may vary from one implementation to another to fulfill some special needs.

4.2.3 Spack Shutdown Hook

Along with the utility script it is necessary to create a hook inside the entry point, to catch the calling arguments of the program. This allows Spack to redirect them to the jailed environment. After the arguments got redirected, the hook should ensure that the main distribution does not execute the given command. This can be easily prevented by stopping Spack, after the command got executed inside the jail. See listing 2 for the implementation details.

```python
# check sys.argv[1] against isolate allow the call to
# isolate --remove-environment without
# being trapped inside a chroot jail
if isolate and sys.argv[1] != 'isolate':
    # check if spack is inside the isolated environment
    if spack_root != "/home/spack":
        # if that is not the case create the jail
        # with the helper function
        isolate_environment()
    # exit the main process because
    # it was not intend to be called
    sys.exit(0)
```

Listing 2: Example how to shutdown the environment after calling the Spack instance inside the jail.

The hook, as shown in listing 2, is directly implemented inside the __init__.py file after the general initialization. This avoids the execution of the given command by the unjailed Spack instance.

4.2.4 Command Line

Spack comes already with an command called bootstrap.

```
$ ./spack bootstrap
```

But this command is restricted to clone a new repository of Spack into a new directory, along with some parameters to specify a git branch. Therefore it may be useful to extend the command, to execute the desired behavior. Because this command is already occupied, it should be easier to create a new command to avoid the confusion of users. Hence the implementation should provide functions to generate and destroy the jail. As well as create a shell inside the client, to support advanced commands such as

```
$ ./spack isolate --build-environment path
```

to create a new jail at the specified path. The isolate parameter then allows to keep the old implementation without interference with the bootstrap command.

Additionally it may be useful to provide a command line interface to destroy the generated environment via

```
$ ./spack isolate --remove-environment
```

to clean up all mounted files or remove the virtual machine.

```
$ ./spack isolate --cli
```

To create a local shell inside the generated environment.

This should provide all necessary functions in order to work with the new feature.

4.3 Whitelisting

Pros	Cons
• Easy to implement	• Require administrator rights
• Total control over all files	• Each necessary file must be whitelisted
	• Reuses the files of the host operating system

The simplest approach to generate a jailed environment is by blend out all files, which are not part of the necessary system files. This also includes files, which are required to compile a simple program. Additionally to the system files there might be the need to keep the basic system utilities or some programs which are required to run Spack (for example Python).

Branch features/bootstrap-whitelisting

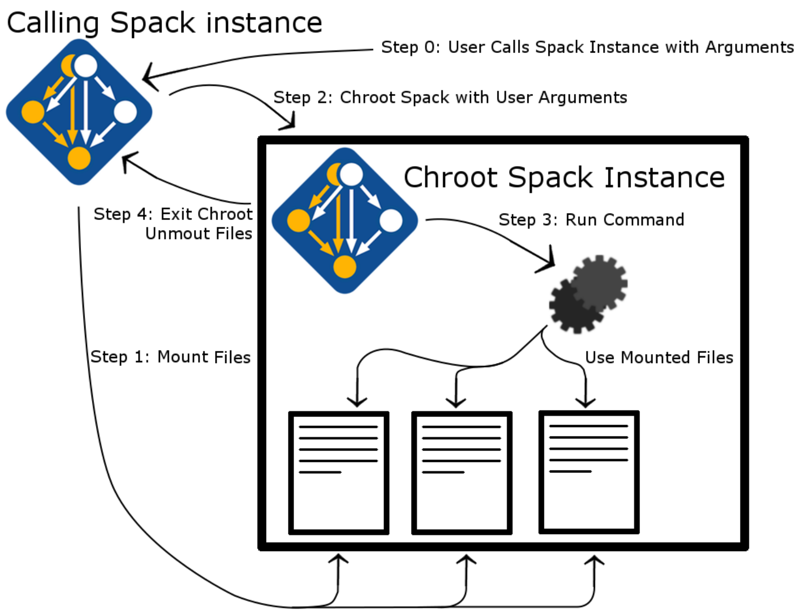
Calling Spack instance
Step 0: User Calls Spack Instance with Arguments
Step 2: Chroot Spack with User Arguments
Chroot Spack Instance
Step 3: Run Command
Step 4: Exit Chroot Unmout Files
Step 1: Mount Files
Use Mounted Files

Figure 2: General overview, how to jail Spack in a chroot environment. Spack Logo taken from `https://computation.llnl.gov/sites/default/files/public/styles/project_logo/adaptive-image/public/spack-logo-220-LLNL.png?itok=yFLw1dWN`.

This can be done by listing all files within a whitelist, for the specified system. While this is a time consuming task, it still has its benefits. As for example to specify the exact files, which should be used by the compiler. Another advantage is the simple implementation, where it is only necessary to mount bind all the whitelisted files to a specified directory. After the binding process it is then necessary to create the chroot environment. The only required extra work is to copy an implementation of Spack into the generated environment, before the generation of the jail. This allows passing the program arguments to a Spack instance inside the jail, without rewriting the entire Spack implementation or writing a client. The generation of a Spack instance can be also simplified, by calling Spack from the shell implementation, inside the jail. The shell itself get called from the outside Spack instance (see figure 2). But it has also its cons, as mentioned before, it is necessary to list all require files to call the compiler, this would include the binutils, Python, perl, all system includes and a compiler. Additionally to the includes, another requirement for the user is to acquire administrator rights to create the jail.

4.4 Distribution Package Manager

Pros	Cons
• Easy to implement	• Require administrator rights
• Total control over all packages	• No control over which files get included into the whitelist, without a blacklist
• Dependencies get resolved automatically	• Reuses the files of the host operating system
	• Depends on the package manager of the host system

A way to avoid the tedious work of the whitelist generation, is by using the package manager of the distribution to generate the whitelist. Because the development distribution uses the dpkg package manager, the implementation focused on it. But it should also be possible to switch to another package manager, such as yum or pacman. This reduces the whitelist to a list, which contains only the name packages, which are required to run Spack and compile a program with a compiler.

To find all dependencies, it is necessary to list all available packages on the host system by executing:

```
$ dpkg -l
```

(see [Ker17a] and especially [Ker17c]). This list all packages which are currently installed. It lists the short name, a long name, the architecture and a description of the package (see table 1).

```
ii   vim-runtime     2:7.4.1689-3ubuntu1.2      all      Vi IMproved - Runtime files
ii   wget            1.17.1-1ubuntu1.2          amd64    retrieves files from the web
ii   xauth           1:1.0.9-1ubuntu2           amd64    X authentication utility
ii   xfsprogs        4.3.0+nmu1ubuntu1          amd64    Utilities for managing the XFS filesystem
ii   xz-utils        5.1.1alpha+20120614-2u     amd64    XZ-format compression utilities
ii   zlib1g:amd64    1:1.2.8.dfsg-2ubuntu4.     amd64    compression library - runtime
```

Table 1: Example output of dpkg -l

Therefore it is necessary to check the list against the whitelist; To identify if the required package is installed. If all packages are available, it is mandatory

Branch features/bootstrap-packagesystem

to find all dependencies of each package through:

```
$ apt-cache depends
```

(see [Ker17a] and especially [Nie17a]). This results in a list of dependencies and recommendations, as well as conflicts (see table 2).

```
wget
  Depends: libc6
  Depends: libidn11
  Depends: libpcre3
  Depends: libssl1.0.0
  Depends: libuuid1
  Depends: zlib1g
  Conflicts: <wget-ssl>
  Recommends: ca-certificates
```

Table 2: Example output of apt-cache depends wget

This list can then be used to generate the chroot jail as in the whitelist approach (see chapter 4.3). While this method reduces the amount of work to generate the whitelist, it is still required to manage a smaller version of it. Therefore it allows fine control about the packages, which should be included into the generated whitelist. At the same time, by introducing the whitelist on a package level, it results in the loss of control at the file level. This can be prevented with a blacklist, which may result in the same amount of work as with the whitelist approach. Further this control does not resolve the problems of the whitelist implementation, as for example the required administrator rights or the reusage of the host files. Additionally it results in a huge amount of files, which need to be bound. Hence these files may not be required to compile a program or run Spack. This may result in a long time to bootstrap the environment, because every single file must be bound by a single mount bind call. Therefore the implementation does not result in any advantages over the whitelist approach other than the reduced whitelist.

4.5 Systemimages

Pros	Cons
• Control over installed packages	• Some distributions require a extensive preconfiguration
• Simple usage, by using an already generated distributions	• Currently only work with tar files
• Only one file to mount required, which can be mounted permanently in fstab	
• No administrator rights required	

Another way to avoid the usage of the host system files and the generation of a whitelist is by using an existing distribution. This allows for the creation of a distribution with all required packages, through existing programs such as mkosi[6]. Alternatively it is possible to use a preexisting configuration from the distributor of the distribution or by using an OpenStack image[7]. While some distribution require a extensive preconfiguration, such as Arch Linux, most distributions should work out of the box.

A restriction to use a system image of a Linux distribution, is to provide a standard compatible dictionary structure by providing the /dev, /sys and /proc (see [Wir04] especially chapter 3) directories and the resolv.conf file. This also applies to the host system as well. If that is the case, it is only necessary to copy all required files to the assigned location and mount bind the /dev, /sys and /proc directory, to provide all required devices in the jail. Further it is required to copy the /etc/resolv.conf to provide the name lookup system for the jail as well. Another minor restriction of the current implementation is the constraint to provide the system files as a tar file. Using tar allows to provide the least common denominator for all possible systems, as a result it should be available on each system. Because there

Branch features/bootstrap-systemimages

Branch features/bootstrap-final

[6]`https://github.com/systemd/mkosi`

[7]`https://docs.openstack.org/image-guide/obtain-images.html`

a only a few files, which are required to mount, they are ideally for writing them into the fstab file. This allows to keep the created environment permanently even after restarting the system without acquire administrator privileges. Another approach to solve the mount bind privilege problem is by creating a daemon (see [Pro05]), which has the right to bind /dev, /sys and /proc to the desired location. This has the advantage to be available for all users without the need to be an administrator. Additionally the daemon have to be created only once by the administrator or the startup process before the usage of Spack. This allows to track all mount bind calls to avoid mounting or unmounting a directory twice. This approach may also be useful for the previous implementations, but it also introduces some security vulnerabilities. Because the whitelist approach could mount any file on the host, the daemon must support this behavior. Thereby allowing every user to call the daemon to bind any specified folder to another location. This minimizes the system security, because it allows jailed users to get out of the jail by creating a mount point to the root directory of the host. By restricting the daemon to mount the /dev, /sys and /proc directories, it minimizes this security problem.

While this only prevents the capability problem of mount, this does not solve the demand for administrator rights of the chroot command. Fortunately the Linux version 3.8 introduces the command unshare (see [Ker17a] and especially [Ker17j]). Unshare allows to fork the namespaces of the root process to a child process. Notably it allows to unshare the network, mount and process id namespaces. Thereby it allows to fork another shell which has root rights inside its own user namespace. This enables the possibility to acquired root rights inside its own namespace. With this right it is then possible to run the chroot command without the need to be a real root user.

Additionally to the method of unsharing a namespace and create the mount points through a daemon, it is possible to use the namespace system directly. Because unshare prevents the execution of setuid and setgid [Ker17j] it is not possible to drop privileges. This happens to be a problem, when using some applications which try to remap the user id. As for example tar, when decompressing a tarball. Because unshare only allows to be root inside the generated namespace or keep the current user. Therefore, when generating the chroot environment through unshare, it is necessary to map the generated user as root, to gain the necessary privileges CAP_SYS_ADMIN and CAP_SYS_CHROOT. But after the generation of the chroot environment the privileges should be dropped. While this is not possible with unshare, it could be achieved by using namespaces [Ker17f], [Ker17k], [Ker17g], [Ker17e],

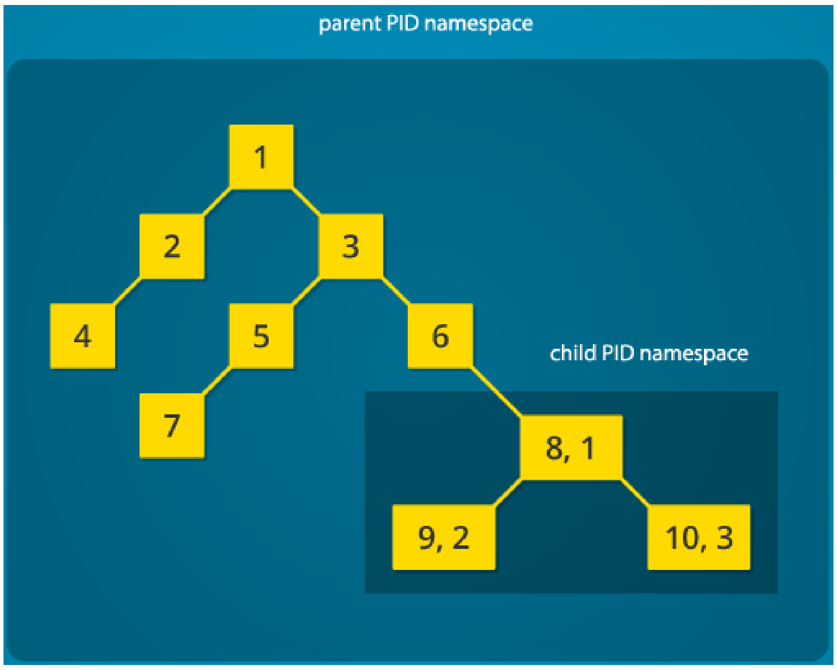

Figure 3: Example for the pid namespace. The parent sees the whole tree, while the child has only access to its own namespace. The process ids also get mapped automatically inside the child namespace.
Source: [Rid17]

[Rid17] directly. This approach is actually very similar to using a container system. When using this system it separates the resource from the root namespace (as show in figure 3 and 4). When using a namespace, the child is only allowed to view the resources in its own namespace. This allows to generate a user namespace with the necessary capabilities. Therefore it allows to run chroot and mount bind while being user. To achieve this, it is necessary to clone the current process with root rights inside its newly generated namespace. When calling clone it is possible to assign the cloned process a new namespace for the user id [Ker17k], process id [Ker17g], network, etc. After this step, it is important to mount a temporary file system (tmpfs) as the new root file system for the process, while keeping the user filesystem in a sub directory. Through the generation of another directory inside this tmpfs root directory, it is then possible to mount bind the files in the actual user directory to the newly generated directory.

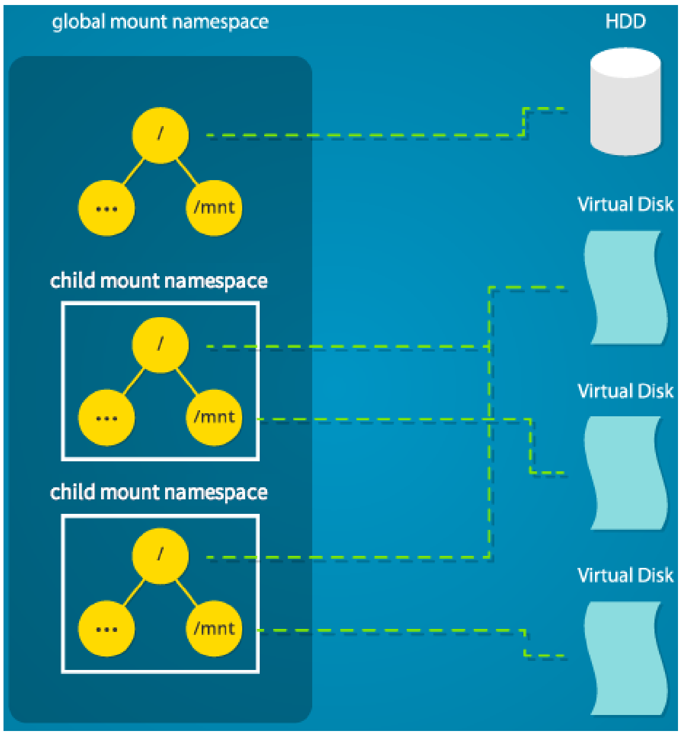

Figure 4: Example for the mount namespaces. The parent has access to the actual disk, while the child has only access to its own namespace. Source: [Rid17]

This allows to mount bind all necessary files. After these steps are done, it is important to chroot the cloned child process to the generated directory, which contains the mount points. Therefore the process can only see the files which are mounted inside the tmpfs. Then it is possible to drop all privileges, to become non root. This allows to keep the current user and group id inside the jail and therefore to execute the Spack instance as the current user.

4.6 Container System

With the recent up rise of container systems such as Docker[8], LinuX Container (LXC)[9] or Rkt[10], the Linux environment introduced different methods to run a program in an predefined environment, without the need to virtualize them through a virtual machine. Unfortunately the current state of the

Branch features/bootstrap-container

[8]https://www.docker.com/

[9]https://linuxcontainers.org/lxc/introduction/

[10]https://github.com/rkt/rkt

container systems is not that stable.

The reason to use a container system to run Spack, is the simplified architecture without the need for an virtual machine. This is possible by using the newly introduces Linux namespaces. These allow to unshare some parts of a child process (see the explanation about unshare in the previous chapter 4.5). They also have the benefit to support a better isolation of an arbitrary process, without the loss of performance.

The philosophy of the Rkt project is to mark a used container as garbage. All garbage collected containers get then destroyed (see `https://coreos.com/rkt/docs/latest/devel/pod-lifecycle.html` for more details). While this philosophy has its benefits, it interferes with the goal to use the container as a storage for Spack. Because if Spack got installed inside the container and was used to build the required programs, it is impossible to reuse these programs after the container got garbage collected. Therefore Rkt is not a fitting solution.

An alternative solution could be Docker, but because the development team currently switches to a community / enterprise model (at the time of this work), it is currently impossible to track which features get included into the community edition in future releases.

Another tool to support container systems could be the LXC project, which is also part of the base of Docker. This allows to create containers via namespaces and through an advanced chroot implementation. Unfortunately it requires administrator privileges to run or create a container. While this can be prevented by changing the configuration of LXC (see `https://linuxcontainers.org/lxc/getting-started/`), it restricts the container. Thereby when using an unprivileged container, LXC is not allowed to mount filesystems, create device nodes or do any user / group id operation which is not mapped by the system (see `https://linuxcontainers.org/lxc/getting-started/`).

Because of the restrictions in all container implementations, it is currently not possible to create an implementation within a container, which aligns perfectly with Spack.

4.7 Virtual Machine

Pros	Cons
• Allows every distribution which supports Spack • Multiple architectures possible • Easy installation with virt-install through an ISO file • No administrator rights required • Access from any system possible, because of an ssh client	• Slower than the other approaches • Needs an ssh server on the virtual machine for communication between the two Spack instances • Compiled binaries must run inside the virtual machine • Introduces additional dependencies like libvirt, Qemu, virtinst • Requires additional disk space to simulate the hard drive

The most flexible solution in terms of the support for different architectures and operating systems is a virtual machine. Because there is a huge variety of virtual machines available, it might be reasonable to use libvirt[11] as an abstraction layer. Along with virt-install[12] it allows easily to create a virtual machine by specifying the operating system through [Nie17b].

```
$ virt-install --virt-type=kvm --ram=512mb \
               --vcpu=1 --crom=ubuntu.iso
```

The benefit of this approach is the simplified implementation, which only consist out of a wrapper for the tools. Therefore it is only necessary to create the virtual environment through the given tools. Additionally it is important to install an ssh client on the client system. This allows the communication between the host and the client without any additional software, even over the system boundary. The downside of this implementation is the introduction of a hypervisor, which results in a small loss of performance. While this might be acceptable with modern systems, it is a huge obstacle when it comes to HPC. Because Spack relies on rpaths to run the selected application it is not

Branch features/bootstrap-vm
[11]https://libvirt.org/
[12]https://linux.die.net/man/1/virt-install

possible to copy the compiled programs to the host system without mimicking the directory structure of the client system. Therefore it is necessary to run the compiled binary inside the virtual machine, which is problematic if the binary should perform on multiple clusters as fast as possible. If the host hardware does not support virtualization, the performance of the system might be even to slow to compile a program. Another requirement for the virtual machine is the need to have storage to simulate the hard drive. This usually is larger than the operating system itself and its size depend on the users need.

5 Conclusion

While the best fitting implementation depends on the needs of the user, it is clear that the whitelisting approaches have no benefit over the system image approach. Therefore the best choice is between the system image (see chapter 4.5) or the virtual machine implementation (see chapter 4.7). Because Spack is designed for the HPC environment, it might be more suitable to use the system image approach. This allows the choice of the operating system without losing performance. As a fallback it might be possible to run the environment inside a virtual machine, to support a different architecture than the host system. As for now each implementation has its own branch. Therefore it is not possible to combine the system image with the virtual machine implementation, but this might change in the future. The implementations can be found at `https://github.com/TheTimmy/spack.git` under the branch prefix specified in the footnote of each implementation chapter.

References

[Gam15] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. “The Spack Package Manager: Bringing Order to HPC Software Chaos”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: ACM, 2015, 40:1–40:12. ISBN: 978-1-4503-3723-6. DOI: `10.1145/2807591.2807623`. URL: `http://doi.acm.org/10.1145/2807591.2807623` (cit. on p. 2).

[Gam16] Todd Gamblin, Gregory Becker, Greg Lee, Matt Legendre, Massimiliano Culpo, Benedikt Hegner, and Elizabeth Fischer. “Managing HPC Software Complexity with Spack”. In: (2016). URL: `http://llnl.github.io/spack/files/Spack-SC16-Tutorial.pdf` (cit. on pp. 2, 3).

[Gam17a] Todd Gamblin. “Packaging Guide”. In: (2017). URL: `http://spack.readthedocs.io/en/latest/packaging_guide.html` (visited on 09/23/2017) (cit. on p. 6).

[Gam17b] Todd Gamblin. “Spack”. In: (2017). URL: `http://spack.readthedocs.io/en/latest/` (visited on 09/23/2017) (cit. on pp. 2, 4–6).

[Gam17c] Todd Gamblin. “spack install”. In: (2017). URL: `http://spack.readthedocs.io/en/latest/basic_usage.html#cmd-spack-install` (visited on 09/23/2017) (cit. on p. 4).

[Gam17d] Todd Gamblin. “spack.build_systems package”. In: (2017). URL: `http://spack.readthedocs.io/en/latest/spack.build_systems.html` (visited on 09/23/2017) (cit. on p. 2).

[JM05] Gough Brian J. and Stallman Richard M. “An Introduction to GCC - for the GNU compilers gcc and g++”. In: (2005). ISSN: ISBN 0954161793. URL: `http://www.network-theory.co.uk/docs/gccintro/gccintro_21.html` (visited on 09/23/2017) (cit. on p. 7).

[Ker17a] Michael et al. Kerrisk. “The Linux man-pages project”. In: (2017). URL: `https://www.kernel.org/doc/man-pages/` (visited on 09/23/2017) (cit. on pp. 8, 9, 14, 15, 17).

[Ker17b] Michael et al. Kerrisk. “The Linux man-pages project, CHROOT(2)”. In: (2017). URL: `http://man7.org/linux/man-pages/man2/chroot.2.html` (visited on 09/23/2017) (cit. on p. 8).

[Ker17c] Michael et al. Kerrisk. “The Linux man-pages project, dpkg(1)”. In: (2017). URL: `http://man7.org/linux/man-pages/man1/dpkg.1.html` (visited on 09/23/2017) (cit. on p. 14).

[Ker17d] Michael et al. Kerrisk. “The Linux man-pages project, MOUNT(2)”. In: (2017). URL: `http://man7.org/linux/man-pages/man2/mount.2.html` (visited on 09/23/2017) (cit. on p. 9).

[Ker17e] Michael et al. Kerrisk. “The Linux man-pages project, MOUNT_NAMESPACES(7)”. In: (2017). URL: `http://man7.org/linux/man-pages/man7/mount_namespaces.7.html` (visited on 09/23/2017) (cit. on p. 17).

[Ker17f] Michael et al. Kerrisk. “The Linux man-pages project, NAMESPACES(7)”. In: (2017). URL: `http://man7.org/linux/man-pages/man7/namespaces.7.html` (visited on 09/23/2017) (cit. on p. 17).

[Ker17g] Michael et al. Kerrisk. “The Linux man-pages project, PID_NAMESPACES(7)”. In: (2017). URL: `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html` (visited on 09/23/2017) (cit. on pp. 17, 18).

[Ker17h] Michael et al. Kerrisk. “The Linux man-pages project, RANDOM(4)”. In: (2017). URL: `http://man7.org/linux/man-pages/man4/random.4.html` (visited on 09/23/2017) (cit. on p. 9).

[Ker17i] Michael et al. Kerrisk. “The Linux man-pages project, SYSTEMD-NSPAWN(1)”. In: (2017). URL: `http://man7.org/linux/man-pages/man1/systemd-nspawn.1.html` (visited on 09/23/2017) (cit. on p. 8).

[Ker17j] Michael et al. Kerrisk. “The Linux man-pages project, UNSHARE(1)”. In: (2017). URL: `http://man7.org/linux/man-pages/man1/unshare.1.html` (visited on 09/23/2017) (cit. on pp. 8, 17).

[Ker17k] Michael et al. Kerrisk. “The Linux man-pages project, USER_NAMESPACES(7)”. In: (2017). URL: `http://man7.org/linux/man-pages/man7/user_namespaces.7.html` (visited on 09/23/2017) (cit. on pp. 17, 18).

[Nie17a] Gustavo et al. Niemeyer. “apt-cache(8) - Linux man page”. In: (2017). URL: `https://linux.die.net/man/8/apt-cache` (visited on 09/23/2017) (cit. on p. 15).

[Nie17b] Gustavo et al. Niemeyer. “virt-install(1) - Linux man page”. In: (2017). URL: `https://linux.die.net/man/1/virt-install` (visited on 09/23/2017) (cit. on p. 21).

[Pro05] The Linux Information Project. “Daemon Definition”. In: (2005). URL: `http://www.linfo.org/daemon.html` (visited on 09/23/2017) (cit. on p. 17).

[Rid17] Mahmud Ridwan. “Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces”. In: (2017). URL: `https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces` (visited on 09/23/2017) (cit. on pp. 18, 19).

[Wir04] Lars Wirzenius, Joanna Oja, Stephen Stafford, and Alex Weeks. “The Linux System Administrator’s Guide”. In: (2004). URL: `http://www.tldp.org/LDP/sag/html/index.html` (visited on 09/23/2017) (cit. on p. 16).