Fs-utils: File Systems Access Tools for Userland

Arnaud Ysmal (stacktic@NetBSD.org) and Antti Kantee (pooka@cs.hut.fi)

September 2009

Abstract

Currently, file system access is done either through a kernel driver or a specialized userspace program. In the former case the file system is mounted with the mount utility and then accessed with standard userspace tools such as cat. An example of the latter case is the mtools utility suite which allows to access msdos file systems.

The NetBSD Runnable Userspace Meta Program (RUMP) framework enables the use of kernel file system drivers as part of userspace programs. By building upon RUMP and NetBSD base system utilities such as ls and cp, we have created the fs-utils application suite. It provides mtools-like file system access without requiring mount privileges or an in-kernel driver. In contrast to mtools, fs-utils reuses the kernel file system drivers instead of relying on a userspace reimplementation, supports a total of 12 file systems from NetBSD plus FUSE file systems, and offers the same usage as the well-known tools (e.g. all of the flags of ls are supported). In addition to running on NetBSD, there has been initial success in running fs-utils on Linux.

This paper explains how these programs were written and recounts the evolution of the project. It also shows the benefits and use-cases of fs-utils.

1 Introduction

A file system is a mechanism that allows a user to organize his or hers data with files and directories. It enables to read a file “foo” in the directory “bar” instead of having to remember the block and sector of the data.

A file system image contains the file system’s internal representation of the file system’s state. A file system image can either be a regular file or it can be contained on a mass media partition. Additionally, in the case of NFS or another networked file system, we consider the connection to the server to be the file system image.

File systems can be accessed by kernel drivers (what we do when using mount) or by userland tools. In this paper, we focus on the latter, or more precisely on userland tools using the file system code from the NetBSD kernel. This principle is implemented on NetBSD and known as RUMP (Runnable Userspace Meta Program) [1]. We can also use NetBSD kernel code in userspace on non-NetBSD operating systems, as we show later in this paper.

The rest of this paper is organized as follows. The remainder of this Section will present fs-utils and explain what we wanted to achieve by creating this new set of tools. Section 2 focuses on the way it was developed, discover possible use cases and point out its benefits. Section 3 analyses our approach and Section 4 presents some common use cases. Section 5 presents some recent developments and finally Section 6 concludes.

1.1 What is fs-utils?

fs-utils is a set of utilities (including commands like cat, chflags, chmod, cp, du, find, ln, ls, mkdir, mkfifo, mv, rm, rmdir, touch) that give access to a file system image. It was developed on top of RUMP and UKFS libraries. With this, it is possible to use commands like:

```
$ fsu_ls ffs.img /etc/defaults
daily.conf      pf.boot.conf    security.conf
monthly.conf    rc.conf         weekly.conf
$ fsu_cat ffs.img /etc/defaults/weekly.conf
$ echo "foo" | fsu_write ffs.img  bar
$ fsu_cat ffs.img bar
foo
```

1.2 Why fs-utils?

The aim of this project is to provide a set of tools to manipulate file systems, like mtools[2], e2fsprogs[3].

Mtools opens the possibility of accessing and/or modifying FAT file systems by offering a set of commands. For instance, it is possible to use the command mdir to list a directory or mren to rename a file.

fs-utils wants to achieve the same main goal. However, unlike mtools, the purpose of fs-utils is to provide a generic set of commands which could be used on many different file systems instead of focusing on a particular one.

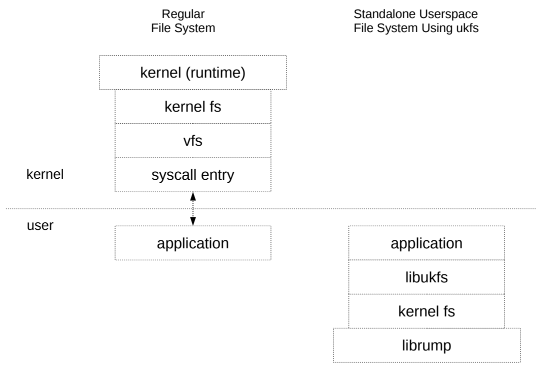
That is why fs-utils has been designed to handle every file system supported by RUMP.

With this set of tools, anyone can manipulate a file system image he has access to without having to mount it. Mounting is, by default, an operation only allowed to root user. However with this type of access, there is no need to be root.

Possible operations depend on the file system image permissions. The user will be able to read file contents and browse directories with read access and modify the files contained in the image if he or she has write permissions to the image. Permissions of files inside the image do not have any effect as the user running a utility from fs-utils is considered root for operations on the file system image.

1.3 RUMP and UKFS

fs-utils was developed on top of the UKFS library (User-Kernel File System Library) [4] and RUMP (Runnable Userspace Meta Programs) [5].



UKFS is a library which uses RUMP. The UKFS interface is based on pathnames and each call is self-contained for ease of use. This means that it is not necessary to e.g. open a file before reading it. The file system image is mounted internally within the process using UKFS. The kernel mount system call is not used, and the mountpoint is accessible only within the process which is did the mounting.

The following is an example of a program using UKFS. It mounts an ffs image, creates a directory called “dir” and reads 11 bytes from a file. The file system is then unmounted.

```
fs = ukfs_mount("ffs", "/ffs.img", "/", 0,
    &args, sizeof(args));
ukfs_mkdir(fs, "/dir", 0777);
ukfs_read(fs, "/a/file", 0, buf, 11);
ukfs_release(fs, 0);
```

2 Development

2.1 Origin of fs-utils

fs-utils started as a NetBSD project for the Google Summer of Code 2008[6]. It was written as a proof that file systems servers can work in userland.

2.2 Development

The first step was to start writing some tools to get in touch with the UKFS library and see what should be added in it to ease its usage.

Few functions were added, mostly about files status like changing the owner, setting permissions and getting information. These commands were needed due to their use in a large number of commands that were planned.

The attempt was to write commands completely from scratch. This created problems in terms of maintainability since a new set of commands had to be maintained, although it was quite similar to the standards commands.

While writing these commands, some functions appeared to be generic enough to be shared in libraries. In these functions, we can find file handling and directory opening and reading and virtually mounting and unmounting. It was clear that we needed at least two libraries, the first one, called fsu_mount, takes care of the mounting and unmounting and the second one, called fsu_utils, replaces functions from the libc that access file systems.

After writing both libraries, and doing some experimentation, the decision to base future versions on existing utility code was made. This means that fs-utils mkdir would use the same code as the base system mkdir, and so forth.

2.2.1 fsu_mount

fsu_mount takes care of everything needed to mount a file system. It is able to test all supported file systems until it finds one that match the file system used for the current device. The algorithm used to do this is quite simple. It tries to mount the image with each supported file system, one by one, until it find a file system which is able to parse the command line arguments and mount returns successfully. With this, the user does not have to give the file system type of the image.

Code needed to mount a file system image with this library:

```
#include <fsu_mount.h>

DECLARE_UKFS(ukfs)

int main(int argc, char *argv[])
{
    ...
    FSU_MOUNT(argc, argv, ukfs);
    ...
}
```

`DECLARE_UKFS(ukfs)` means that the program will use a struct ukfs which is called ukfs. Then `FSU_MOUNT(argc, argv, ukfs);` will use the standard command line arguments to process the mount. For example, the arguments "-o ro /dev/wd1a" might be given. This would result in a read-only mount of /dev/wd1a with the mount handle stored in "ukfs".

2.2.2 fsu_utils

fsu_utils is a library which takes care of three aspects.

- The first one is implementing functions from the libc that manage file operations to pass through UKFS, like opening, reading, writing and closing. All of these functions are prefixed with "fsu_", for instance: fsu_fopen, fsu_fread, etc...
- The second is replacing function that manage directories like opening and reading a directory. They are also prefixed with "fsu_".
- The last one is fts(3). It is a set of functions which allows the user to recursively get the files and/or directories within a directory. It is used by programs like ls, chmod, chown, etc... A modified version of fts(3) was written so as to use the code that exists and is well tested. By doing this, we were able to use most of the code from ls, chmod and so on with less modifications as possible.

2.2.3 Commands

The remaining work was to modify classic programs to use these libraries. These modifications consist in:

- Adding a ukfs declaration (`DECLARE_UKFS(ukfs)`) and call to the virtual mount (`FSU_MOUNT(argc, argv, ukfs)`), provided by the fsu_mount library.
- Replacing the calls to libc functions to their counterparts in the fsu_utils library or in ukfs:

```
#define stat(path, sb) ukfs_stat(ukfs, path, sb)
#define lstat(path, sb) ukfs_lstat(ukfs, path, sb)
```

- Updating the usage to contain the file system image path argument

These changes were added to the code using ifdef so as to be able to compile the programs with file system access through RUMP or through libc and system calls.

So far, updates of these programs are done manually.

2.2.4 Installing

As of now, there are two ways of installing these programs for NetBSD 5.0. The first one is to use the package available in pkgsrc[7] under filesystems/fs-utils. pkgsrc is the NetBSD packages collection framework. The second one, which is the version with the newest code, consists in getting the code directly from the NetBSD othersrc cvs repository and compiling it. The former will install the binaries in your pkgbase/bin; the latter will install the binaries in /usr/local/bin.

3 Benefits

3.1 Analysis of mtools-like approaches

GNU mtools[2] is a set of tools to access/modify a FAT file system. It is composed of tools prefixed with a m like mcat, mdu, mren, ... Their approach consists in entirely rewriting the code of the commands and focusing on one specific file system.

There are two main drawbacks in this approach.

The first one is that the code for the file system has to be written from scratch. The new code to handle file systems can contains bugs as this code is not as well tested as existing implementations which have been in use for a long time, such as kernel drivers.

While these programs have the main functionality of their POSIX counterparts (mcat and cat, mdu and du), they do not follow the entire specification (for instance, not all flags are supported).

The other disadvantage is that, in general, these programs support only one file system. This means that you need to have one program for each command and for each file system you want to access this way. We can see examples in Minix, with mtools for accessing FAT file systems, isoinfo, isodir and isoread for accessing ISO9660 file systems and ext2tools[8] for accessing ext2fs. In these cases, a different set of tools must be used for each file system, which also means a different usage for the same command.

3.2 Analysis of our approach

3.2.1 Our approach

Our approach consists in trying to avoid writing totally new code, so far as we can, in order to reduce the effort needed to maintain it. That is why we decided to reuse code from the NetBSD kernel. Furthermore, since the code comes from the very base of the system, it is therefore well tested.

In the NetBSD kernel, the file system section consists in two parts : generic file system handling and specific code for each file system.

RUMP reuses them to create a generic library called rumpvfs and a specific library for each file system supported in the kernel.

As the kernel code is used in a way than is not usual, it could be possible to detect new bugs in it and fix them in the NetBSD kernel code.

3.2.2 Benefits of RUMP

RUMP also allows to improve maintainability. The code of each file system is held in a library which can be updated (rebuild) with patches added to the file system code in the kernel since the compilation of these programs. This means that rebuilding the RUMP libraries is enough to get the benefits of a newly added patch in the kernel code.

Each file system is held in a library called librumpfs_<fs>. These libraries are loaded dynamically so that we can support new file systems without recompiling the tools.

3.2.3 Portability

A clear advantage of userspace reimplementations such as mtools is that the same codebase works on multiple operating systems. For fs-utils to be a viable alternative, it must be usable on operating systems beyond NetBSD. This requires running NetBSD kernel code in userspace on other operating systems.

We investigated running fs-utils on a i386 Gentoo Linux machine and found that we could compile RUMP, UKFS and fs-utils on Linux and successfully access a file system image. However, support is in a very early stage and it will require a lot more work before fs-utils is ready for public use on non-NetBSD machines. The major problem is that Linux and NetBSD ABIs do not match: struct stat, struct timeval and struct dirent are of different binary layout. For fs-utils to be fully functional on other operating systems, translation to and from the NetBSD ABI needs to be written. Since translation is limited to the structures mentioned above, we argue that this is a reasonable task even for supporting all the major operating systems.

3.2.4 Available commands

There are three kind of tools: the standard tools, the fs-utils specific tools and the console tool. They can also be seen as in-image tools, tools to transfer files from/to the image to the host file system and a mini shell that uses previous commands.

The standards POSIX tools which were adapted to use RUMP are: chflags, chmod, chown, cp, du, ln, ls, mkdir, mkfifo, mknod, rm, rmdir.

```
$ fsu_mkdir ffs.img /foo
$ fsu_ls ffs.img -ld /foo
drwxr-xr-x 2 root wsrc 512 Aug 30 17:02 /foo
$ fsu_chmod ffs.img 740 /foo
$ fsu_chown ffs.img stacktic:users /foo
$ fsu_ls ffs.img -ld /foo
drwxr----- 2 stacktic users  512 Aug 30 17:02 /foo
$ fsu_mkfifo ffs.img /foo/pipe
$ fsu_mknod ffs.img /foo/block  b 0
$ fsu_cp ffs.img -R /foo /bar
$ fsu_ls ffs.img -l /bar
brw-r--r-- 1 root wsrc 0, 0 Aug 30 17:11 block
prw-r--r-- 1 root wsrc    0 Aug 30 17:11 pipe
$ fsu_rm ffs.img -r /bar
$ fsu_ls ffs.img /bar
fsu_ls: /bar: No such file or directory
```

The following tools had to be rewritten: cat, diff, find, mv, touch.

```
$ fsu_cat
usage: fsu_cat [-o mnt_args] [[-t] fstype]
       [-f] fsdevice [-benstv] [-] filename
```

```
$ fsu_cat ffs.img /txtfile
foo bar
$ fsu_mv ffs.img /txtfile /txtfile2
$ fsu_cat ffs.img /txtfile2
foo bar
$ fsu_diff /txtfile /txtfile2 && echo same
same
```

3.2.5 fs-utils specific tools

These tools were written specifically for fs-utils:

- fsu_console: command line tool acting as a shell to help browsing the image

  ```
  $ fsu_console ffs.img
  ffs.img(ffs):/ # ls
  altroot boot.cfg etc libdata mnt    root stand usr
  bin     dev      lib libexec rescue sbin tmp   var
  ffs.img(ffs):/ # cd /root
  ffs.img(ffs):/root # ls -l .shrc
  -rw-r--r-- 1 root wheel 221 Aug 29 13:38 .shrc
  ffs.img(ffs):/root # chmod u+x .shrc
  ffs.img(ffs):/root # ls -l .shrc
  -rwxr--r-- 1 root wheel 221 Aug 29 13:38 .shrc
  ```

- fsu_ecp: command allowing the user to transfer files from/to the image to/from the host file system (it is also called fsu_put and fsu_get). fsu_put is also used to import files to an image by a new version of makefs[1]

- fsu_exec: command allowing the user to execute a local program on a file contained in the image. For instance, "fsu_exec an_image $EDITOR foo" enables you to edit the file named foo contained on the image an_image with your favorite editor (which is not on the image)

- fsu_write: command writing its input to a file in the image (`ls foo | fsu_write image /bar` will write the output of the ls foo to a file called bar in the image root directory)

3.2.6 Supported file systems

At the very time of writing this paper, this set of tools can interact with 12 file systems (iso9660, efs, ext2fs, ffs, hfs, lfs, msdos, nfs, ntfs, sysvbfs, tmpfs, udf) plus file systems written for fuse[9], thanks to the refuse[10] library. It has been successfully tested on the fuse-ntfs-3g file system, which is available on pkgsrc[7] in filesystems/fuse-ntfs-3g.

Moreover, it was written so as to facilitate the addition of new file systems. It is not necessary to recompile the commands to support new file systems.

3.3 Benefits of userland file system server

- Security as the file system code runs in userland
- Allows users to manipulate their own images without having to mount them with a system command
- Easier file system code debugging
- Easier file system code development

This kind of access avoids problems due to malformed file systems which can lead to kernel crashes. Instead of a kernel crash, there will only be a crash of the application and the generation of a core dump which helps debugging to debug the file system implementation.

4 Use cases

In these use cases, the term mount can have two meanings, the first one is mount as a kernel mount which is the standard way of mounting. The second meaning is a rump mount or per-process mounting which is a user space mount.

- The first use case we thought about was the access to a file system image, so that any one owning an image with read and/or write permission on it can access the image without having to mount it. Thanks to this, it is possible to manipulate file system images without having to be root and having to use vnconfig and mount but also without any special option in the kernel.

- It also works for non-image based file systems. As an example, the Network File System is supported by RUMP. For example, it allows a non-root user to get/put files from/to a NFS server he has access to, with the fsu_ecp command.

- A user can also use these tools on a real block device he has access to, even if the current kernel was not built with the support for the corresponding file system or if he does not trust the partition, which can be the case for removable devices like USB sticks.

- It is possible to use these tools to make file system development easier. While developing a generic file system mounting protocol this year, we were able to use these programs to test the code written some minutes before without having to reboot or load a module that could panic the kernel (in this project, using module was not possible). After recompiling RUMP libraries, it was possible to do a: `fsu_ls -t ffs ffs.img` (print the list of files contained in the ffs image named ffs.img). As said before, if the application crashed due to a bug, running gdb was possible to help fixing the bug that gave the error. The only bug encountered that was not found while using RUMP was a memory space error (saying that a chunk of memory is in user space when it is in kernel space). This is understandable as everything in RUMP is in user space. Without this memory space issue, it was possible to run the code written in a real kernel without doing changes.

- These tools can also be used by the command line console tools (fsu_console). It makes file system browsing easier.

These commands allow a user to access file systems which are not supported by the kernel or are in development. In the latter case, it can avoid a crash of the whole system. It also provides access to file systems in a secure way that does not compromise the system if the file system has been corrupted - intentionally or not - and that could have crashed the kernel. These commands add privileges to non-root user (browsing their own images) without decreasing the system security.

5 Evolutions since NetBSD 5.0

In this section, we are taking NetBSD 5.0 as reference. This is the first stable version on which RUMP and UKFS were integrated.

5.1 The RUMP pseudo system call way

Since NetBSD 5.0, the implementation of RUMP evolved and is still evolving. The idea of RUMP pseudo system call was added. It allows the user to use functionality equivalent to the real system calls without going to the kernel space. Most file systems related system calls are supported.

A pseudo system call is a function which is not a real system call. The function catching this call is in userland, as opposed to real system calls where the handler is on the kernel side. With these system calls we can get even closer from original programs. This is due to the fact that we do not have to handle a struct representing the virtual mount. Instead of writing : `ukfs_chdir(ukfs, path);`, we use `rump_sys_chdir(path);`. The removal of this struct allows the function to have the same prototype as the system call (in this case `int chdir(const char *path);`).

With this, programs that use file systems access can use two arrays of functions. The first one with real system calls and the second one with RUMP system calls, so that we can choose when we start the program which one we want to use.

Doing so, the UKFS library is no longer needed as the RUMP system calls can do the same. Using these pseudo-system calls makes it easier to handle both access types in the same compiled command. For example, adding RUMP support to ls with this method consists of a unified diff of less than 160 lines (33 lines added and 10 lines changed). The size of original ls is 1692 lines.

6 Conclusions and Future Work

The fs-utils project and its evolution were presented in depth. The main benefits provided by this set of tools are:

- Support for several file systems
- No specific kernel option needed
- Allows users to manipulate their own images without having to mount them with a system command
- Security as the file system code runs in userland
- Use of robust code from the NetBSD kernel
- Easier file system code development
- Easier file system code debugging

One idea of what can be done in the near future is to merge commands from the base system and commands from fs-utils, so as to be able to use a switch to easily choose whether to use the command through RUMP or through standard system calls. As a proof of concept, ls has been implemented. The user can use ls normally or set an environment variable to use ls through RUMP system calls.

The fsu_utils library is composed of rewritten functions of the libc. It could be a good idea to use code from the libc to do this.

The fsu_mount library needs to have specific code for each supported file system. A project that will lead to the removal of this is in development. This project consists in developing a unified mounting protocol. Its aim is to avoid the use of file system specific code during the mounting process.

We could consider installing tools from fs-utils as a part of the base system. It could also be possible to merge the fs-utils utilities and the POSIX ones. To do so, we would have to choose how to use RUMP instead of standard libc/system calls for file system access. It would require the RUMP system calls which were added since NetBSD 5.0.

References

[1] Antti Kantee. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference*, 2009.

[2] GNU mtools . http://www.gnu.org/software/mtools/intro.html.

[3] Theodore Ts'o. E2fsprogs: Ext2/3/4 Filesystem Utilities. http://e2fsprogs.sourceforge.net/, 2008.

[4] Antti Kantee. User-Kernel File System Library (UKFS). http://www.NetBSD.org/docs/rump/ukfs.html.

[5] Antti Kantee. Runnable Userspace Meta Programs. http://www.NetBSD.org/docs/rump/.

[6] Arnaud Ysmal. File system access utilities (fs-utils). http://www.NetBSD.org/∼stacktic/fs-utils.html.

[7] pkgsrc: The NetBSD Packages Collection. http://www.pkgsrc.org.

[8] Terry McConnell. ext2 file system access tools. http://barnyard.syr.edu/software.shtml.

[9] Filesystem in Userspace. http://fuse.sourceforge.net.

[10] A. Kantee and A. Crooks. ReFUSE: Userspace FUSE Reimplementation Using puffs. In *EuroBSDCon*, 2007.