Some aspects of Unix file-system security

Markus Wenzel
TU München

October 1, 2005

Abstract

Unix is a simple but powerful system where everything is either a process or a file. Access to system resources works mainly via the file-system, including special files and devices. Most Unix security issues are reflected directly within the file-system. We give a mathematical model of the main aspects of the Unix file-system including its security model, but ignoring processes. Within this formal model we discuss some aspects of Unix security, including a few odd effects caused by the general "worse-is-better" approach followed in Unix.

Our formal specifications will be giving in simply-typed classical set-theory as provided by Isabelle/HOL. Formal proofs are expressed in a human-readable fashion using the structured proof language of Isabelle/Isar, which is a system intended to support intelligible semi-automated reasoning over a wide range of application domains. Thus the present development also demonstrates that Isabelle/Isar is sufficiently flexible to cover typical abstract verification tasks as well. So far this has been the classical domain of interactive theorem proving systems based on unstructured tactic languages.

Contents

1 Introduction

1.1 The Unix philosophy

Over the last 2 or 3 decades the Unix community has collected a certain amount of folklore wisdom on building systems that actually work, see [6] for further historical background information. Here is a recent account of the philosophical principles behind the Unix way of software and systems engineering.[1]

```
The UNIX Philosophy (Score:2, Insightful)
by yebb on Saturday March 25, @11:06AM EST (#69)
(User Info)

The philosophy is a result of more than twenty years of software
development and has grown from the UNIX community instead of being
enforced upon it. It is a defacto-style of software development. The
nine major tenets of the UNIX Philosophy are:

  1. small is beautiful
  2. make each program do one thing well
  3. build a prototype as soon as possible
  4. choose portability over efficiency
  5. store numerical data in flat files
  6. use software leverage to your advantage
  7. use shell scripts to increase leverage and portability
  8. avoid captive user interfaces
  9. make every program a filter

The Ten Lesser Tenets

  1. allow the user to tailor the environment
  2. make operating system kernels small and lightweight
  3. use lower case and keep it short
  4. save trees
  5. silence is golden
  6. think parallel
  7. the sum of the parts if greater than the whole
  8. look for the ninety percent solution
  9. worse is better
 10. think hierarchically
```

The "worse-is-better" approach quoted above is particularly interesting. It basically means that *relevant* concepts have to be implemented in the right way, while *irrelevant* issues are simply ignored in order to avoid unnecessary complication of the design and implementation. Certainly, the overall

[1] This has appeared on *Slashdot* on 25-March-2000, see http://slashdot.com.

quality of the resulting system heavily depends on the virtue of distinction between the two categories of "relevant" and "irrelevant".

1.2 Unix security

The main entities of a Unix system are *files* and *processes* [4]. Files subsume any persistent "static" entity managed by the system — ranging from plain files and directories, to more special ones such device nodes, pipes etc. On the other hand, processes are "dynamic" entities that may perform certain operations while being run by the system.

The security model of classic Unix systems is centered around the file system. The operations permitted by a process that is run by a certain user are determined from information stored within the file system. This includes any kind of access control, such as read/write access to some plain file, or read-only access to a certain global device node etc. Thus proper arrangement of the main Unix file-system is very critical for overall security.[2]

Generally speaking, the Unix security model is a very simplistic one. The original designers did not have maximum security in mind, but wanted to get a decent system working for typical multi-user environments. Contemporary Unix implementations still follow the basic security model of the original versions from the early 1970's [6]. Even back then there would have been better approaches available, albeit with more complexity involved both for implementers and users.

On the other hand, even in the 2000's many computer systems are run with little or no file-system security at all, even though virtually any system is exposed to the net in one way or the other. Even "personal" computer systems have long left the comfortable home environment and entered the wilderness of the open net sphere.

This treatment of file-system security is a typical example of the "worse-is-better" principle introduced above. The simplistic security model of Unix got widely accepted within a large user community, while the more innovative (and cumbersome) ones are only used very reluctantly and even tend to be disabled by default in order to avoid confusion of beginners.

1.3 Odd effects

Simplistic systems usually work very well in typical situations, but tend to exhibit some odd features in non-typical ones. As far as Unix file-system security is concerned, there are many such features that are well-known to experts, but may surprise naive users.

[2]Incidently, this is why the operation of mounting new volumes into the existing file space is usually restricted to the super-user.

Subsequently, we consider an example that is not so exotic after all. As may be easily experienced on a running Unix system, the following sequence of commands may put a user's file-system into an uncouth state. Below we assume that `user1` and `user2` are working within the same directory (e.g. somewhere within the home of `user1`).

```
user1> umask 000; mkdir foo; umask 022
user2> mkdir foo/bar
user2> touch foo/bar/baz
```

That is, `user1` creates a directory that is writable for everyone, and `user2` puts there a non-empty directory without write-access for others.

In this situation it has become impossible for `user1` to remove his very own directory `foo` without the cooperation of either `user2`, since `foo` contains another non-empty and non-writable directory, which cannot be removed.

```
user1> rmdir foo
rmdir: directory "foo": Directory not empty
user1> rmdir foo/bar
rmdir: directory "bar": Directory not empty
user1> rm foo/bar/baz
rm not removed: Permission denied
```

Only after `user2` has cleaned up his directory `bar`, is `user1` enabled to remove both `foo/bar` and `foo`. Alternatively `user2` could remove `foo/bar` as well. In the unfortunate case that `user2` does not cooperate or is presently unavailable, `user1` would have to find the super user (`root`) to clean up the situation. In Unix `root` may perform any file-system operation without any access control limitations.[3]

Is there really no other way out for `user1` in the above situation? Experiments can only show possible ways, but never demonstrate the absence of other means exhaustively. This is a typical situation where (formal) proof may help. Subsequently, we model the main aspects Unix file-system security within Isabelle/HOL [3] and prove that there is indeed no way for `user1` to get rid of his directory `foo` without help by others (see §5.4 for the main theorem stating this).

The formal techniques employed in this development are the typical ones for abstract "verification" tasks, namely induction and case analysis over the structure of file-systems and possible system transitions. Isabelle/HOL

[3]This is the typical Unix way of handling abnormal situations: while it is easy to run into odd cases due to simplistic policies it is as well quite easy to get out. There are other well-known systems that make it somewhat harder to get into a fix, but almost impossible to get out again!

[3] is particularly well-suited for this kind of application. By the present development we also demonstrate that the Isabelle/Isar environment [7, 8] for readable formal proofs is sufficiently flexible to cover non-trivial verification tasks as well. So far this has been the classical domain of "interactive" theorem proving systems based on unstructured tactic languages.

2 Unix file-systems

theory *Unix*
imports *Nested-Environment List-Prefix*
begin

We give a simple mathematical model of the basic structures underlying the Unix file-system, together with a few fundamental operations that could be imagined to be performed internally by the Unix kernel. This forms the basis for the set of Unix system-calls to be introduced later (see §3), which are the actual interface offered to processes running in user-space.

Basically, any Unix file is either a *plain file* or a *directory*, consisting of some *content* plus *attributes*. The content of a plain file is plain text. The content of a directory is a mapping from names to further files.[4] Attributes include information to control various ways to access the file (read, write etc.).

Our model will be quite liberal in omitting excessive detail that is easily seen to be "irrelevant" for the aspects of Unix file-systems to be discussed here. First of all, we ignore character and block special files, pipes, sockets, hard links, symbolic links, and mount points.

2.1 Names

User ids and file name components shall be represented by natural numbers (without loss of generality). We do not bother about encoding of actual names (e.g. strings), nor a mapping between user names and user ids as would be present in a reality.

types
uid = nat
name = nat
path = name list

2.2 Attributes

Unix file attributes mainly consist of *owner* information and a number of *permission* bits which control access for "user", "group", and "others" (see

[4]In fact, this is the only way that names get associated with files. In Unix files do *not* have a name in itself. Even more, any number of names may be associated with the very same file due to *hard links* (although this is excluded from our model).

the Unix man pages *chmod(2)* and *stat(2)* for more details).

Our model of file permissions only considers the "others" part. The "user" field may be omitted without loss of overall generality, since the owner is usually able to change it anyway by performing *chmod*.[5] We omit "group" permissions as a genuine simplification as we just do not intend to discuss a model of multiple groups and group membership, but pretend that everyone is member of a single global group.[6]

datatype *perm* =
  Readable
| *Writable*
| *Executable* — (ignored)

types *perms* = *perm set*

record *att* =
  owner :: *uid*
  others :: *perms*

For plain files *Readable* and *Writable* specify read and write access to the actual content, i.e. the string of text stored here. For directories *Readable* determines if the set of entry names may be accessed, and *Writable* controls the ability to create or delete any entries (both plain files or sub-directories).

As another simplification, we ignore the *Executable* permission altogether. In reality it would indicate executable plain files (also known as "binaries"), or control actual lookup of directory entries (recall that mere directory browsing is controlled via *Readable*). Note that the latter means that in order to perform any file-system operation whatsoever, all directories encountered on the path would have to grant *Executable*. We ignore this detail and pretend that all directories give *Executable* permission to anybody.

2.3 Files

In order to model the general tree structure of a Unix file-system we use the arbitrarily branching datatype $('a, 'b, 'c)$ *env* from the standard library of Isabelle/HOL [1]. This type provides constructors *Val* and *Env* as follows:

$$Val :: 'a \Rightarrow ('a, 'b, 'c)\ env$$
$$Env :: 'b \Rightarrow ('c \Rightarrow ('a, 'b, 'c)\ env\ option) \Rightarrow ('a, 'b, 'c)\ env$$

Here the parameter $'a$ refers to plain values occurring at leaf positions, parameter $'b$ to information kept with inner branch nodes, and parameter

[5] The inclined Unix expert may try to figure out some exotic arrangements of a real-world Unix file-system such that the owner of a file is unable to apply the *chmod* system call.

[6] A general HOL model of user group structures and related issues is given in [2].

$'c$ to the branching type of the tree structure. For our purpose we use the type instance with $att \times string$ (representing plain files), *att* (for attributes of directory nodes), and *name* (for the index type of directory nodes).

types

$file = (att \times string,\ att,\ name)\ env$

The HOL library also provides *lookup* and *update* operations for general tree structures with the subsequent primitive recursive characterizations.

$lookup :: ('a,\ 'b,\ 'c)\ env \Rightarrow 'c\ list \Rightarrow ('a,\ 'b,\ 'c)\ env\ option$
$update :: 'c\ list \Rightarrow ('a,\ 'b,\ 'c)\ env\ option \Rightarrow ('a,\ 'b,\ 'c)\ env \Rightarrow ('a,\ 'b,\ 'c)\ env$

```
lookup env xs =
(case xs of [] ⇒ Some env
 | x # xs ⇒
     case env of Val a ⇒ None
     | Env b es ⇒ case es x of None ⇒ None | Some e ⇒ lookup e xs)
```

```
update xs opt env =
(case xs of [] ⇒ case opt of None ⇒ env | Some e ⇒ e
 | x # xs ⇒
     case env of Val a ⇒ Val a
     | Env b es ⇒
         case xs of [] ⇒ Env b (es(x := opt))
         | y # ys ⇒
             Env b
              (es(x := case es x of None ⇒ None
                          | Some e ⇒ Some (update (y # ys) opt e))))
```

Several further properties of these operations are proven in [1]. These will be routinely used later on without further notice.

Apparently, the elements of type *file* contain an *att* component in either case. We now define a few auxiliary operations to manipulate this field uniformly, following the conventions for record types in Isabelle/HOL [3].

constdefs

```
attributes :: file ⇒ att
attributes file ≡
  (case file of
     Val (att, text) ⇒ att
  | Env att dir ⇒ att)
```

```
attributes-update :: att ⇒ file ⇒ file
attributes-update att file ≡
  (case file of
     Val (att', text) ⇒ Val (att, text)
  | Env att' dir ⇒ Env att dir)
```

lemma [*simp*]: *attributes* (*Val* (*att*, *text*)) = *att*
  by (*simp add*: *attributes-def*)

lemma [*simp*]: *attributes* (*Env att dir*) = *att*
  by (*simp add*: *attributes-def*)

lemma [*simp*]: *attributes* (*file* (|*attributes* := *att*|)) = *att*
  by (*cases file*) (*simp-all add*: *attributes-def attributes-update-def*
    split-tupled-all)

lemma [*simp*]: (*Val* (*att*, *text*)) (|*attributes* := *att′*|) = *Val* (*att′*, *text*)
  by (*simp add*: *attributes-update-def*)

lemma [*simp*]: (*Env att dir*) (|*attributes* := *att′*|) = *Env att′ dir*
  by (*simp add*: *attributes-update-def*)

2.4 Initial file-systems

Given a set of *known users* a file-system shall be initialized by providing an empty home directory for each user, with read-only access for everyone else. (Note that we may directly use the user id as home directory name, since both types have been identified.) Certainly, the very root directory is owned by the super user (who has user id 0).

constdefs
  init :: *uid set* ⇒ *file*
  init users ≡
    Env (|*owner* = *0*, *others* = {*Readable*}|)
     (λ*u*. *if u* ∈ *users then Some* (*Env* (|*owner* = *u*, *others* = {*Readable*}|) *empty*)
       else None)

2.5 Accessing file-systems

The main internal file-system operation is access of a file by a user, requesting a certain set of permissions. The resulting *file option* indicates if a file had been present at the corresponding *path* and if access was granted according to the permissions recorded within the file-system.

Note that by the rules of Unix file-system security (e.g. [4]) both the superuser and owner may always access a file unconditionally (in our simplified model).

constdefs
  access :: *file* ⇒ *path* ⇒ *uid* ⇒ *perms* ⇒ *file option*
  access root path uid perms ≡
    (*case lookup root path of*
      None ⇒ *None*
    | *Some file* ⇒
        if uid = *0*

$\vee$ *uid* = *owner* (*attributes file*)
$\vee$ *perms* $\subseteq$ *others* (*attributes file*)
then Some file
else None)

Successful access to a certain file is the main prerequisite for system-calls to be applicable (cf. §3). Any modification of the file-system is then performed using the basic *update* operation.

We see that *access* is just a wrapper for the basic *lookup* function, with additional checking of attributes. Subsequently we establish a few auxiliary facts that stem from the primitive *lookup* used within *access*.

lemma *access-empty-lookup*: *access root path uid* {} = *lookup root path*
by (*simp add*: *access-def split*: *option.splits*)

lemma *access-some-lookup*:
access root path uid perms = *Some file* $\Longrightarrow$
lookup root path = *Some file*
by (*simp add*: *access-def split*: *option.splits if-splits*)

lemma *access-update-other*: *path'* $\parallel$ *path* $\Longrightarrow$
access (*update path' opt root*) *path uid perms* = *access root path uid perms*
proof −
assume *path'* $\parallel$ *path*
then obtain *y z xs ys zs* **where**
$y \neq z$ **and** *path'* = *xs* @ *y* # *ys* **and** *path* = *xs* @ *z* # *zs*
by (*blast dest*: *parallel-decomp*)
hence *lookup* (*update path' opt root*) *path* = *lookup root path*
by (*blast intro*: *lookup-update-other*)
thus *?thesis* **by** (*simp only*: *access-def*)
qed

3 File-system transitions

3.1 Unix system calls

According to established operating system design (cf. [4]) user space processes may only initiate system operations by a fixed set of *system-calls*. This enables the kernel to enforce certain security policies in the first place.[7]

In our model of Unix we give a fixed datatype *operation* for the syntax of system-calls, together with an inductive definition of file-system state transitions of the form *root* −*x*→ *root'* for the operational semantics.

datatype *operation* =

[7] Incidently, this is the very same principle employed by any "LCF-style" theorem proving system according to Milner's principle of "correctness by construction", such as Isabelle/HOL itself.

Read uid string path
| *Write uid string path*
| *Chmod uid perms path*
| *Creat uid perms path*
| *Unlink uid path*
| *Mkdir uid perms path*
| *Rmdir uid path*
| *Readdir uid name set path*

The *uid* field of an operation corresponds to the *effective user id* of the underlying process, although our model never mentions processes explicitly. The other parameters are provided as arguments by the caller; the *path* one is common to all kinds of system-calls.

consts
uid-of :: *operation* ⇒ *uid*
primrec
uid-of (*Read uid text path*) = *uid*
uid-of (*Write uid text path*) = *uid*
uid-of (*Chmod uid perms path*) = *uid*
uid-of (*Creat uid perms path*) = *uid*
uid-of (*Unlink uid path*) = *uid*
uid-of (*Mkdir uid path perms*) = *uid*
uid-of (*Rmdir uid path*) = *uid*
uid-of (*Readdir uid names path*) = *uid*

consts
path-of :: *operation* ⇒ *path*
primrec
path-of (*Read uid text path*) = *path*
path-of (*Write uid text path*) = *path*
path-of (*Chmod uid perms path*) = *path*
path-of (*Creat uid perms path*) = *path*
path-of (*Unlink uid path*) = *path*
path-of (*Mkdir uid perms path*) = *path*
path-of (*Rmdir uid path*) = *path*
path-of (*Readdir uid names path*) = *path*

Note that we have omitted explicit *Open* and *Close* operations, pretending that *Read* and *Write* would already take care of this behind the scenes. Thus we have basically treated actual sequences of real system-calls *open*–*read*/*write*–*close* as atomic.

In principle, this could make big a difference in a model with explicit concurrent processes. On the other hand, even on a real Unix system the exact scheduling of concurrent *open* and *close* operations does *not* directly affect the success of corresponding *read* or *write*. Unix allows several processes to have files opened at the same time, even for writing! Certainly, the result from reading the contents later may be hard to predict, but the system-calls involved here will succeed in any case.

The operational semantics of system calls is now specified via transitions of the file-system configuration. This is expressed as an inductive relation (although there is no actual recursion involved here).

consts
  transition :: (*file* × *operation* × *file*) *set*

syntax
  -transition :: *file* ⇒ *operation* ⇒ *file* ⇒ *bool*
  (- −-→ - [*90*, *1000*, *90*] *100*)
translations
  root −*x*→ *root′* ⇌ (*root*, *x*, *root′*) ∈ *transition*

inductive *transition*
  intros

```
  read:
    access root path uid {Readable} = Some (Val (att, text)) ⟹
      root −(Read uid text path)→ root
  write:
    access root path uid {Writable} = Some (Val (att, text′)) ⟹
      root −(Write uid text path)→ update path (Some (Val (att, text))) root

  chmod:
    access root path uid {} = Some file ⟹
      uid = 0 ∨ uid = owner (attributes file) ⟹
      root −(Chmod uid perms path)→ update path
        (Some (file (|attributes := attributes file (|others := perms|)|))) root

  creat:
    path = parent-path @ [name] ⟹
      access root parent-path uid {Writable} = Some (Env att parent) ⟹
      access root path uid {} = None ⟹
      root −(Creat uid perms path)→ update path
        (Some (Val ((|owner = uid, others = perms|), []))) root
  unlink:
    path = parent-path @ [name] ⟹
      access root parent-path uid {Writable} = Some (Env att parent) ⟹
      access root path uid {} = Some (Val plain) ⟹
      root −(Unlink uid path)→ update path None root

  mkdir:
    path = parent-path @ [name] ⟹
      access root parent-path uid {Writable} = Some (Env att parent) ⟹
      access root path uid {} = None ⟹
      root −(Mkdir uid perms path)→ update path
        (Some (Env (|owner = uid, others = perms|) empty)) root
  rmdir:
```

```
  path = parent-path @ [name] ⟹
    access root parent-path uid {Writable} = Some (Env att parent) ⟹
    access root path uid {} = Some (Env att' empty) ⟹
    root −(Rmdir uid path)→ update path None root

readdir:
  access root path uid {Readable} = Some (Env att dir) ⟹
    names = dom dir ⟹
    root −(Readdir uid names path)→ root
```

Certainly, the above specification is central to the whole formal development. Any of the results to be established later on are only meaningful to the outside world if this transition system provides an adequate model of real Unix systems. This kind of "reality-check" of a formal model is the well-known problem of *validation*.

If in doubt, one may consider to compare our definition with the informal specifications given the corresponding Unix man pages, or even peek at an actual implementation such as [5]. Another common way to gain confidence into the formal model is to run simple simulations (see §4.2), and check the results with that of experiments performed on a real Unix system.

3.2 Basic properties of single transitions

The transition system *root −x→ root'* defined above determines a unique result *root'* from given *root* and x (this is holds rather trivially, since there is even only one clause for each operation). This uniqueness statement will simplify our subsequent development to some extent, since we only have to reason about a partial function rather than a general relation.

```
theorem transition-uniq: root −x→ root' ⟹ root −x→ root'' ⟹ root' = root''
proof −
  assume root: root −x→ root'
  assume root −x→ root''
  thus root' = root''
  proof cases
    case read
    with root show ?thesis by cases auto
  next
    case write
    with root show ?thesis by cases auto
  next
    case chmod
    with root show ?thesis by cases auto
  next
    case creat
    with root show ?thesis by cases auto
  next
```

case *unlink*
with *root* **show** *?thesis* **by** *cases auto*
next
case *mkdir*
with *root* **show** *?thesis* **by** *cases auto*
next
case *rmdir*
with *root* **show** *?thesis* **by** *cases auto*
next
case *readdir*
with *root* **show** *?thesis* **by** *cases fastsimp+*
qed
qed

Apparently, file-system transitions are *type-safe* in the sense that the result of transforming an actual directory yields again a directory.

theorem *transition-type-safe*:
$root\ {-x\rightarrow}\ root' \Longrightarrow \exists\, att\ dir.\ root = Env\ att\ dir$
$\Longrightarrow \exists\, att\ dir.\ root' = Env\ att\ dir$
proof −
assume *tr*: $root\ {-x\rightarrow}\ root'$
assume *inv*: $\exists\, att\ dir.\ root = Env\ att\ dir$
show *?thesis*
proof (*cases path-of x*)
case *Nil*
with *tr inv* **show** *?thesis*
by *cases* (*auto simp add*: *access-def split*: *if-splits*)
next
case *Cons*
from *tr* **obtain** *opt* **where**
$root' = root \lor root' = update\ (path\text{-}of\ x)\ opt\ root$
by *cases auto*
with *inv Cons* **show** *?thesis*
by (*auto simp add*: *update-eq split*: *list.splits*)
qed
qed

The previous result may be seen as the most basic invariant on the file-system state that is enforced by any proper kernel implementation. So user processes — being bound to the system-call interface — may never mess up a file-system such that the root becomes a plain file instead of a directory, which would be a strange situation indeed.

3.3 Iterated transitions

Iterated system transitions via finite sequences of system operations are modeled inductively as follows. In a sense, this relation describes the cumulative effect of the sequence of system-calls issued by a number of running processes over a finite amount of time.

consts
  transitions :: (*file* × *operation list* × *file*) *set*

syntax
  -transitions :: *file* ⇒ *operation list* ⇒ *file* ⇒ *bool*
  (- =-⇒ - [*90*, *1000*, *90*] *100*)
translations
  root =*xs*⇒ *root*′ ⇌ (*root*, *xs*, *root*′) ∈ *transitions*

inductive *transitions*
  intros
    nil: *root* =[]⇒ *root*
    cons: *root* −*x*→ *root*′ ⟹ *root*′ =*xs*⇒ *root*′′ ⟹ *root* =(*x* # *xs*)⇒ *root*′′

We establish a few basic facts relating iterated transitions with single ones, according to the recursive structure of lists.

lemma *transitions-nil-eq*: *root* =[]⇒ *root*′ = (*root* = *root*′)
proof
  assume *root* =[]⇒ *root*′
  thus *root* = *root*′ **by** *cases simp-all*
next
  assume *root* = *root*′
  thus *root* =[]⇒ *root*′ **by** (*simp only*: *transitions.nil*)
qed

lemma *transitions-cons-eq*:
  root =(*x* # *xs*)⇒ *root*′′ = (∃ *root*′. *root* −*x*→ *root*′ ∧ *root*′ =*xs*⇒ *root*′′)
proof
  assume *root* =(*x* # *xs*)⇒ *root*′′
  thus ∃ *root*′. *root* −*x*→ *root*′ ∧ *root*′ =*xs*⇒ *root*′′
    by *cases auto*
next
  assume ∃ *root*′. *root* −*x*→ *root*′ ∧ *root*′ =*xs*⇒ *root*′′
  thus *root* =(*x* # *xs*)⇒ *root*′′
    by (*blast intro*: *transitions.cons*)
qed

The next two rules show how to "destruct" known transition sequences. Note that the second one actually relies on the uniqueness property of the basic transition system (see §3.2).

lemma *transitions-nilD*: *root* =[]⇒ *root*′ ⟹ *root*′ = *root*
  by (*simp add*: *transitions-nil-eq*)

lemma *transitions-consD*:
  root =(*x* # *xs*)⇒ *root*′′ ⟹ *root* −*x*→ *root*′ ⟹ *root*′ =*xs*⇒ *root*′′
proof −

```
  assume root =(x # xs)⇒ root''
  then obtain r' where r': root −x→ r' and root'': r' =xs⇒ root''
    by cases simp-all
  assume root −x→ root'
  with r' have r' = root' by (rule transition-uniq)
  with root'' show root' =xs⇒ root'' by simp
qed
```

The following fact shows how an invariant Q of single transitions with property P may be transferred to iterated transitions. The proof is rather obvious by rule induction over the definition of *root =xs⇒ root'*.

```
lemma transitions-invariant:
  (⋀r x r'. r −x→ r' ⟹ Q r ⟹  P x ⟹ Q r') ⟹
    root =xs⇒ root' ⟹ Q root ⟹ ∀ x ∈ set xs. P x ⟹ Q root'
proof −
  assume r: ⋀r x r'. r −x→ r' ⟹ Q r ⟹  P x ⟹ Q r'
  assume root =xs⇒ root'
  thus Q root ⟹ (∀ x ∈ set xs. P x) ⟹ Q root' (is PROP ?P root xs root')
  proof (induct root xs root')
    fix root assume Q root
    thus Q root .
  next
    fix root root' root'' and x xs
    assume root': root −x→ root'
    assume hyp: PROP ?P root' xs root''
    assume Q: Q root
    assume P: ∀ x ∈ set (x # xs). P x
    hence P x by simp
    with root' Q have Q': Q root' by (rule r)
    from P have ∀ x ∈ set xs. P x by simp
    with Q' show Q root'' by (rule hyp)
  qed
qed
```

As an example of applying the previous result, we transfer the basic type-safety property (see §3.2) from single transitions to iterated ones, which is a rather obvious result indeed.

```
theorem transitions-type-safe:
  assumes root =xs⇒ root'
    and ∃ att dir. root = Env att dir
  shows ∃ att dir. root' = Env att dir
  using transition-type-safe and prems
proof (rule transitions-invariant)
  show ∀ x ∈ set xs. True by blast
qed
```

4 Executable sequences

An inductively defined relation such as the one of *root* $-x\rightarrow$ *root'* (see §3.1) has two main aspects. First of all, the resulting system admits a certain set of transition rules (introductions) as given in the specification. Furthermore, there is an explicit least-fixed-point construction involved, which results in induction (and case-analysis) rules to eliminate known transitions in an exhaustive manner.

Subsequently, we explore our transition system in an experimental style, mainly using the introduction rules with basic algebraic properties of the underlying structures. The technique closely resembles that of Prolog combined with functional evaluation in a very simple manner.

Just as the "closed-world assumption" is left implicit in Prolog, we do not refer to induction over the whole transition system here. So this is still purely positive reasoning about possible executions; exhaustive reasoning will be employed only later on (see §5), when we shall demonstrate that certain behavior is *not* possible.

4.1 Possible transitions

Rather obviously, a list of system operations can be executed within a certain state if there is a result state reached by an iterated transition.

constdefs
  can-exec :: *file* ⇒ *operation list* ⇒ *bool*
  can-exec root xs ≡ ∃ *root'*. *root* =*xs*⇒ *root'*

lemma *can-exec-nil*: *can-exec root* []
  by (*unfold can-exec-def*) (*blast intro*: *transitions.intros*)

lemma *can-exec-cons*:
    root −*x*→ *root'* ⟹ *can-exec root' xs* ⟹ *can-exec root* (*x* # *xs*)
  by (*unfold can-exec-def*) (*blast intro*: *transitions.intros*)

In case that we already know that a certain sequence can be executed we may destruct it backwardly into individual transitions.

lemma *can-exec-snocD*: ⋀*root*. *can-exec root* (*xs* @ [*y*])
    ⟹ ∃ *root' root''*. *root* =*xs*⇒ *root'* ∧ *root'* −*y*→ *root''*
  (**is** *PROP ?P xs* **is** ⋀*root*. *?A root xs* ⟹ *?C root xs*)
proof (*induct xs*)
  fix *root*
  {
    assume *?A root* []
    thus *?C root* []
      by (*simp add*: *can-exec-def transitions-nil-eq transitions-cons-eq*)

```
  next
    fix x xs
    assume hyp: PROP ?P xs
    assume asm: ?A root (x # xs)
    show ?C root (x # xs)
    proof -
      from asm obtain r root'' where x: root -x→ r and
          xs-y: r =(xs @ [y])⇒ root''
        by (auto simp add: can-exec-def transitions-nil-eq transitions-cons-eq)
      from xs-y hyp obtain root' r' where xs: r =xs⇒ root' and y: root' -y→ r'
        by (unfold can-exec-def) blast
      from x xs have root =(x # xs)⇒ root'
        by (rule transitions.cons)
      with y show ?thesis by blast
    qed
  }
qed
```

4.2 Example executions

We are now ready to perform a few experiments within our formal model of Unix system-calls. The common technique is to alternate introduction rules of the transition system (see §3), and steps to solve any emerging side conditions using algebraic properties of the underlying file-system structures (see §2).

```
lemmas eval = access-def init-def

theorem u ∈ users ⟹ can-exec (init users)
    [Mkdir u perms [u, name]]
  apply (rule can-exec-cons)
    — back-chain can-exec (of Cons)
  apply (rule mkdir)
    — back-chain Mkdir
  apply (force simp add: eval)+
    — solve preconditions of Mkdir
  apply (simp add: eval)
    — peek at resulting dir (optional)


 1. u ∈ users ⟹
    can-exec
     (Env (|owner = 0, others = {Readable}|)
       ((λu. if u ∈ users
             then Some (Env (|owner = u, others = {Readable}|) empty)
             else None)
        (u ↦
         Env (|owner = u, others = {Readable}|)
          [name ↦ Env (|owner = u, others = perms|) empty])))
     []
```

apply (*rule can-exec-nil*)
— back-chain *can-exec* (of *Nil*)
done

By inspecting the result shown just before the final step above, we may gain confidence that our specification of Unix system-calls actually makes sense. Further common errors are usually exhibited when preconditions of transition rules fail unexpectedly.

Here are a few further experiments, using the same techniques as before.

theorem $u \in users \Longrightarrow can\text{-}exec\ (init\ users)$
$[Creat\ u\ perms\ [u,\ name],$
$Unlink\ u\ [u,\ name]]$
apply (*rule can-exec-cons*)
apply (*rule creat*)
apply (*force simp add*: *eval*)+
apply (*simp add*: *eval*)
apply (*rule can-exec-cons*)
apply (*rule unlink*)
apply (*force simp add*: *eval*)+
apply (*simp add*: *eval*)

peek at result:

$$
\begin{array}{l}
1.\ u \in users \Longrightarrow \\
\quad can\text{-}exec \\
\quad (Env\ (\!|owner = 0,\ others = \{Readable\}|\!) \\
\quad\quad ((\lambda u.\ if\ u \in users \\
\quad\quad\quad\quad then\ Some\ (Env\ (\!|owner = u,\ others = \{Readable\}|\!)\ empty) \\
\quad\quad\quad\quad else\ None) \\
\quad\quad\quad (u \mapsto Env\ (\!|owner = u,\ others = \{Readable\}|\!)\ empty))) \\
\quad []
\end{array}
$$

apply (*rule can-exec-nil*)
done

theorem $u \in users \Longrightarrow Writable \in perms_1 \Longrightarrow$
$Readable \in perms_2 \Longrightarrow name_3 \neq name_4 \Longrightarrow$
$can\text{-}exec\ (init\ users)$
$[Mkdir\ u\ perms_1\ [u,\ name_1],$
$Mkdir\ u'\ perms_2\ [u,\ name_1,\ name_2],$
$Creat\ u'\ perms_3\ [u,\ name_1,\ name_2,\ name_3],$
$Creat\ u'\ perms_3\ [u,\ name_1,\ name_2,\ name_4],$
$Readdir\ u\ \{name_3,\ name_4\}\ [u,\ name_1,\ name_2]]$
apply (*rule can-exec-cons*, *rule transition.intros*,
(*force simp add*: *eval*)+, (*simp add*: *eval*)?)+

peek at result:

$1.\ u \in users \Longrightarrow$

$Writable \in perms_1 \Longrightarrow$
$Readable \in perms_2 \Longrightarrow$
$name_3 \neq name_4 \Longrightarrow$
can-exec
$(Env\ (\|owner = 0,\ others = \{Readable\}\|)$
$((\lambda u.\ if\ u \in users$
$then\ Some\ (Env\ (\|owner = u,\ others = \{Readable\}\|)\ empty)$
$else\ None)$
$(u \mapsto$
$Env\ (\|owner = u,\ others = \{Readable\}\|)$
$[name_1 \mapsto$
$Env\ (\|owner = u,\ others = perms_1\|)$
$[name_2 \mapsto$
$Env\ (\|owner = u',\ others = perms_2\|)$
$[name_3 \mapsto Val\ ((\|owner = u',\ others = perms_3\|),\ []),\ name_4 \mapsto$
$Val\ ((\|owner = u',\ others = perms_3\|),\ [])]]])))$
$[]$

apply (*rule can-exec-nil*)
done

theorem $u \in users \Longrightarrow Writable \in perms_1 \Longrightarrow Readable \in perms_3 \Longrightarrow$
can-exec (*init users*)
$[Mkdir\ u\ perms_1\ [u,\ name_1],$
$Mkdir\ u'\ perms_2\ [u,\ name_1,\ name_2],$
$Creat\ u'\ perms_3\ [u,\ name_1,\ name_2,\ name_3],$
$Write\ u'\ ''foo''\ [u,\ name_1,\ name_2,\ name_3],$
$Read\ u\ ''foo''\ [u,\ name_1,\ name_2,\ name_3]]$
apply (*rule can-exec-cons*, *rule transition.intros*,
(*force simp add*: *eval*)+, (*simp add*: *eval*)?)+

peek at result:

1. $u \in users \Longrightarrow$
$Writable \in perms_1 \Longrightarrow$
$Readable \in perms_3 \Longrightarrow$
can-exec
$(Env\ (\|owner = 0,\ others = \{Readable\}\|)$
$((\lambda u.\ if\ u \in users$
$then\ Some\ (Env\ (\|owner = u,\ others = \{Readable\}\|)\ empty)$
$else\ None)$
$(u \mapsto$
$Env\ (\|owner = u,\ others = \{Readable\}\|)$
$[name_1 \mapsto$
$Env\ (\|owner = u,\ others = perms_1\|)$
$[name_2 \mapsto$
$Env\ (\|owner = u',\ others = perms_2\|)$
$[name_3 \mapsto Val\ ((\|owner = u',\ others = perms_3\|),\ ''foo'')]]])))$
$[]$

apply (*rule can-exec-nil*)
done

5 Odd effects — treated formally

We are now ready to give a completely formal treatment of the slightly odd situation discussed in the introduction (see §1): the file-system can easily reach a state where a user is unable to remove his very own directory, because it is still populated by items placed there by another user in an uncouth manner.

5.1 The general procedure

The following theorem expresses the general procedure we are following to achieve the main result.

```
theorem general-procedure:
  (⋀r r'. Q r ⟹ r −y→ r' ⟹ False) ⟹
    (⋀root. init users =bs⇒ root ⟹ Q root) ⟹
    (⋀r x r'. r −x→ r' ⟹ Q r ⟹  P x ⟹ Q r') ⟹
    init users =bs⇒ root ⟹
      ¬ (∃xs. (∀x ∈ set xs. P x) ∧ can-exec root (xs @ [y]))
proof −
  assume cannot-y: ⋀r r'. Q r ⟹ r −y→ r' ⟹ False
  assume init-inv: ⋀root. init users =bs⇒ root ⟹ Q root
  assume preserve-inv: ⋀r x r'. r −x→ r' ⟹ Q r ⟹ P x ⟹ Q r'
  assume init-result: init users =bs⇒ root
  {
    fix xs
    assume Ps: ∀x ∈ set xs. P x
    assume can-exec: can-exec root (xs @ [y])
    then obtain root' root'' where
        xs: root =xs⇒ root' and y: root' −y→ root''
      by (blast dest: can-exec-snocD)
    from init-result have Q root by (rule init-inv)
    from preserve-inv xs this Ps have Q root'
      by (rule transitions-invariant)
    from this y have False by (rule cannot-y)
  }
  thus ?thesis by blast
qed
```

Here $P\ x$ refers to the restriction on file-system operations that are admitted after having reached the critical configuration; according to the problem specification this will become $\mathit{uid\text{-}of}\ x = \mathit{user}_1$ later on. Furthermore, y refers to the operations we claim to be impossible to perform afterwards, we will take *Rmdir* later. Moreover Q is a suitable (auxiliary) invariant over the file-system; choosing Q adequately is very important to make the proof work (see §5.3).

5.2 The particular situation

We introduce a few global declarations and axioms to describe our particular situation considered here. Thus we avoid excessive use of local parameters in the subsequent development.

locale *situation* =
  fixes *users* :: *uid set*
    and $user_1$ :: *uid*
    and $user_2$ :: *uid*
    and $name_1$ :: *name*
    and $name_2$ :: *name*
    and $name_3$ :: *name*
    and $perms_1$ :: *perms*
    and $perms_2$ :: *perms*
  assumes $user_1$*-known*: $user_1 \in users$
    and $user_1$*-not-root*: $user_1 \neq 0$
    and *users-neq*: $user_1 \neq user_2$
    and $perms_1$*-writable*: $Writable \in perms_1$
    and $perms_2$*-not-writable*: $Writable \notin perms_2$
  notes *facts* = $user_1$*-known* $user_1$*-not-root* *users-neq*
    $perms_1$*-writable* $perms_2$*-not-writable*

  fixes *bogus* :: *operation list*
    and *bogus-path* :: *path*
  defines *bogus* ≡
     [*Mkdir* $user_1$ $perms_1$ [$user_1$, $name_1$],
      Mkdir $user_2$ $perms_2$ [$user_1$, $name_1$, $name_2$],
      Creat $user_2$ $perms_2$ [$user_1$, $name_1$, $name_2$, $name_3$]]
    and *bogus-path* ≡ [$user_1$, $name_1$, $name_2$]

The *bogus* operations are the ones that lead into the uncouth situation; *bogus-path* is the key position within the file-system where things go awry.

5.3 Invariance lemmas

The following invariant over the root file-system describes the bogus situation in an abstract manner: located at a certain *path* within the file-system is a non-empty directory that is neither owned and nor writable by $user_1$.

locale *invariant* = *situation* +
  fixes *invariant* :: *file* ⇒ *path* ⇒ *bool*
  defines *invariant root path* ≡
    (∃ *att dir*.
      access root path $user_1$ {} = *Some* (*Env att dir*) ∧ *dir* ≠ *empty* ∧
      $user_1$ ≠ *owner att* ∧

access root path $user_1$ {*Writable*} = *None*)

Following the general procedure (see §5.1) we will now establish the three key lemmas required to yield the final result.

1. The invariant is sufficiently strong to entail the pathological case that $user_1$ is unable to remove the (owned) directory at [$user_1$, $name_1$].

2. The invariant does hold after having executed the *bogus* list of operations (starting with an initial file-system configuration).

3. The invariant is preserved by any file-system operation performed by $user_1$ alone, without any help by other users.

As the invariant appears both as assumptions and conclusions in the course of proof, its formulation is rather critical for the whole development to work out properly. In particular, the third step is very sensitive to the invariant being either too strong or too weak. Moreover the invariant has to be sufficiently abstract, lest the proof become cluttered by confusing detail.

The first two lemmas are technically straight forward — we just have to inspect rather special cases.

lemma (**in** *invariant*)
  cannot-rmdir: *invariant root bogus-path* ⟹
    root −(*Rmdir* $user_1$ [$user_1$, $name_1$])→ *root'* ⟹ *False*
proof −
  assume *invariant root bogus-path*
  then obtain *file* **where** *access root bogus-path* $user_1$ {} = *Some file*
    by (*unfold invariant-def*) *blast*
  moreover
  assume *root* −(*Rmdir* $user_1$ [$user_1$, $name_1$])→ *root'*
  then obtain *att* **where**
      access root [$user_1$, $name_1$] $user_1$ {} = *Some* (*Env att empty*)
    by *cases auto*
  hence *access root* ([$user_1$, $name_1$] @ [$name_2$]) $user_1$ {} = *empty* $name_2$
    by (*simp only*: *access-empty-lookup lookup-append-some*) *simp*
  ultimately show *False* **by** (*simp add*: *bogus-path-def*)
qed

lemma (**in** *invariant*)
  init-invariant: *init users* =*bogus*⇒ *root* ⟹ *invariant root bogus-path*
proof −
  note *eval* = *facts access-def init-def*
  case *rule-context* **thus** *?thesis*
    apply (*unfold bogus-def bogus-path-def*)
    apply (*drule transitions-consD*, *rule transition.intros*,
      (*force simp add*: *eval*)+, (*simp add*: *eval*)*?*)+
      — evaluate all operations

```
    apply (drule transitions-nilD)
      — reach final result
    apply (simp add: invariant-def eval)
      — check the invariant
    done
qed
```

At last we are left with the main effort to show that the "bogosity" invariant is preserved by any file-system operation performed by $user_1$ alone. Note that this holds for any *path* given, the particular *bogus-path* is not required here.

```
lemma (in invariant)
  preserve-invariant: root −x→ root′ ⟹
    invariant root path ⟹ uid-of x = user₁ ⟹ invariant root′ path
proof −
  assume tr: root −x→ root′
  assume inv: invariant root path
  assume uid: uid-of x = user₁

  from inv obtain att dir where
      inv1: access root path user₁ {} = Some (Env att dir) and
      inv2: dir ≠ empty and
      inv3: user₁ ≠ owner att and
      inv4: access root path user₁ {Writable} = None
    by (auto simp add: invariant-def)

  from inv1 have lookup: lookup root path = Some (Env att dir)
    by (simp only: access-empty-lookup)

  from inv1 inv3 inv4 and user₁-not-root
  have not-writable: Writable ∉ others att
    by (auto simp add: access-def split: option.splits if-splits)

  show ?thesis
  proof cases
    assume root′ = root
    with inv show invariant root′ path by (simp only:)
  next
    assume changed: root′ ≠ root
    with tr obtain opt where root′: root′ = update (path-of x) opt root
      by cases auto
    show ?thesis
    proof (rule prefix-cases)
      assume path-of x ∥ path
      with inv root′
      have ⋀perms. access root′ path user₁ perms = access root path user₁ perms
        by (simp only: access-update-other)
      with inv show invariant root′ path
        by (simp only: invariant-def)
```

```
next
  assume path-of x ≤ path
  then obtain ys where path: path = path-of x @ ys ..

  show ?thesis
  proof (cases ys)
    assume ys = []
    with tr uid inv2 inv3 lookup changed path and user_1-not-root
    have False
      by cases (auto simp add: access-empty-lookup dest: access-some-lookup)
    thus ?thesis ..
  next
    fix z zs assume ys: ys = z # zs
    have lookup root' path = lookup root path
    proof −
      from inv2 lookup path ys
      have look: lookup root (path-of x @ z # zs) = Some (Env att dir)
        by (simp only:)
      then obtain att' dir' file' where
          look': lookup root (path-of x) = Some (Env att' dir') and
          dir': dir' z = Some file' and
          file': lookup file' zs = Some (Env att dir)
        by (blast dest: lookup-some-upper)

      from tr uid changed look' dir' obtain att'' where
          look'': lookup root' (path-of x) = Some (Env att'' dir')
        by cases (auto simp add: access-empty-lookup lookup-update-some
          dest: access-some-lookup)
      with dir' file' have lookup root' (path-of x @ z # zs) =
          Some (Env att dir)
        by (simp add: lookup-append-some)
      with look path ys show ?thesis
        by simp
    qed
    with inv show invariant root' path
      by (simp only: invariant-def access-def)
  qed
next
  assume path < path-of x
  then obtain y ys where path: path-of x = path @ y # ys ..

  obtain dir' where
    lookup': lookup root' path = Some (Env att dir') and
    inv2': dir' ≠ empty
  proof (cases ys)
    assume ys = []
    with path have parent: path-of x = path @ [y] by simp
    with tr uid inv4 changed obtain file where
        root' = update (path-of x) (Some file) root
```

```
        by cases auto
      with lookup parent have lookup root′ path = Some (Env att (dir(y↦file)))
        by (simp only: update-append-some update-cons-nil-env)
      moreover have dir(y↦file) ≠ empty by simp
      ultimately show ?thesis ..
    next
      fix z zs assume ys: ys = z # zs
      with lookup root′ path
      have lookup root′ path = Some (update (y # ys) opt (Env att dir))
        by (simp only: update-append-some)
      also obtain file′ where
        update (y # ys) opt (Env att dir) = Env att (dir(y↦file′))
      proof −
        have dir y ≠ None
        proof −
          have dir y = lookup (Env att dir) [y]
            by (simp split: option.splits)
          also from lookup have ... = lookup root (path @ [y])
            by (simp only: lookup-append-some)
          also have ... ≠ None
          proof −
            from ys obtain us u where rev-ys: ys = us @ [u]
              by (cases ys rule: rev-cases) fastsimp+
            with tr path
            have lookup root ((path @ [y]) @ (us @ [u])) ≠ None ∨
                lookup root ((path @ [y]) @ us) ≠ None
              by cases (auto dest: access-some-lookup)
            thus ?thesis by (blast dest!: lookup-some-append)
          qed
          finally show ?thesis .
        qed
        with ys show ?thesis
          by (insert that, auto simp add: update-cons-cons-env)
      qed
      also have dir(y↦file′) ≠ empty by simp
      ultimately show ?thesis ..
    qed

    from lookup′ have inv1′: access root′ path user₁ {} = Some (Env att dir′)
      by (simp only: access-empty-lookup)

    from inv3 lookup′ and not-writable user₁-not-root
    have access root′ path user₁ {Writable} = None
      by (simp add: access-def)
    with inv1′ inv2′ inv3 show ?thesis by (unfold invariant-def) blast
  qed
 qed
qed
```

5.4 Putting it all together

The main result is now imminent, just by composing the three invariance lemmas (see §5.3) according the the overall procedure (see §5.1).

corollary *result*:
 includes *invariant*
 assumes *bogus*: *init users* $=$*bogus*$\Rightarrow$ *root*
 shows $\neg$ ($\exists$ *xs*. ($\forall$ *x* $\in$ *set xs*. *uid-of x* $=$ $user_1$) $\wedge$
  can-exec root (*xs* @ [*Rmdir* $user_1$ [$user_1$, $name_1$]]))
proof –
 from *cannot-rmdir init-invariant preserve-invariant*
  and *bogus* **show** *?thesis* **by** (*rule general-procedure*)
qed

So this is our final result:

$user_1 \in users \Longrightarrow$
$user_1 \neq 0 \Longrightarrow$
$user_1 \neq user_2 \Longrightarrow$
$Writable \in perms_1 \Longrightarrow$
$Writable \notin perms_2 \Longrightarrow$
init
 users =[*Mkdir* $user_1$ $perms_1$ [$user_1$, $name_1$],
   Mkdir $user_2$ $perms_2$ [$user_1$, $name_1$, $name_2$],
   Creat $user_2$ $perms_2$ [$user_1$, $name_1$, $name_2$, $name_3$]]$\Rightarrow$ *root* $\Longrightarrow$
$\neg$ ($\exists$ *xs*. ($\forall$ *x*$\in$*set xs*. *uid-of x* $=$ $user_1$) $\wedge$
   can-exec root (*xs* @ [*Rmdir* $user_1$ [$user_1$, $name_1$]]))

end

References

[1] G. Bauer, T. Nipkow, D. v. Oheimb, L. C. Paulson, T. M. Rasmussen, C. Tabacznyj, and M. Wenzel. The supplemental Isabelle/HOL library. http://isabelle.in.tum.de/library/HOL/Library/document.pdf, 2002.

[2] W. Naraschewski. *Teams as Types — A Formal Treatment of Authorization in Groupware*. PhD thesis, TU München, 2001. Submitted.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*, 2000. http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[4] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[5] L. Torvalds et al. The Linux kernel archives. http://www.kernel.org.

[6] The Unix heritage society. http://minnie.cs.adfa.edu.au/TUHS/.

[7] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

[8] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2002. http://isabelle.in.tum.de/doc/isar-ref.pdf.