

Practical Least Privilege Security

Reader

Appendix B concept 0.5

June 28 2008

MinorFs

Copyright © 2008 Rob J Meijer

rmeijer [at] polacanthus <dot> net

This document holds a concept appendix of the reader for
the training *Practical Least Privilege Security*.

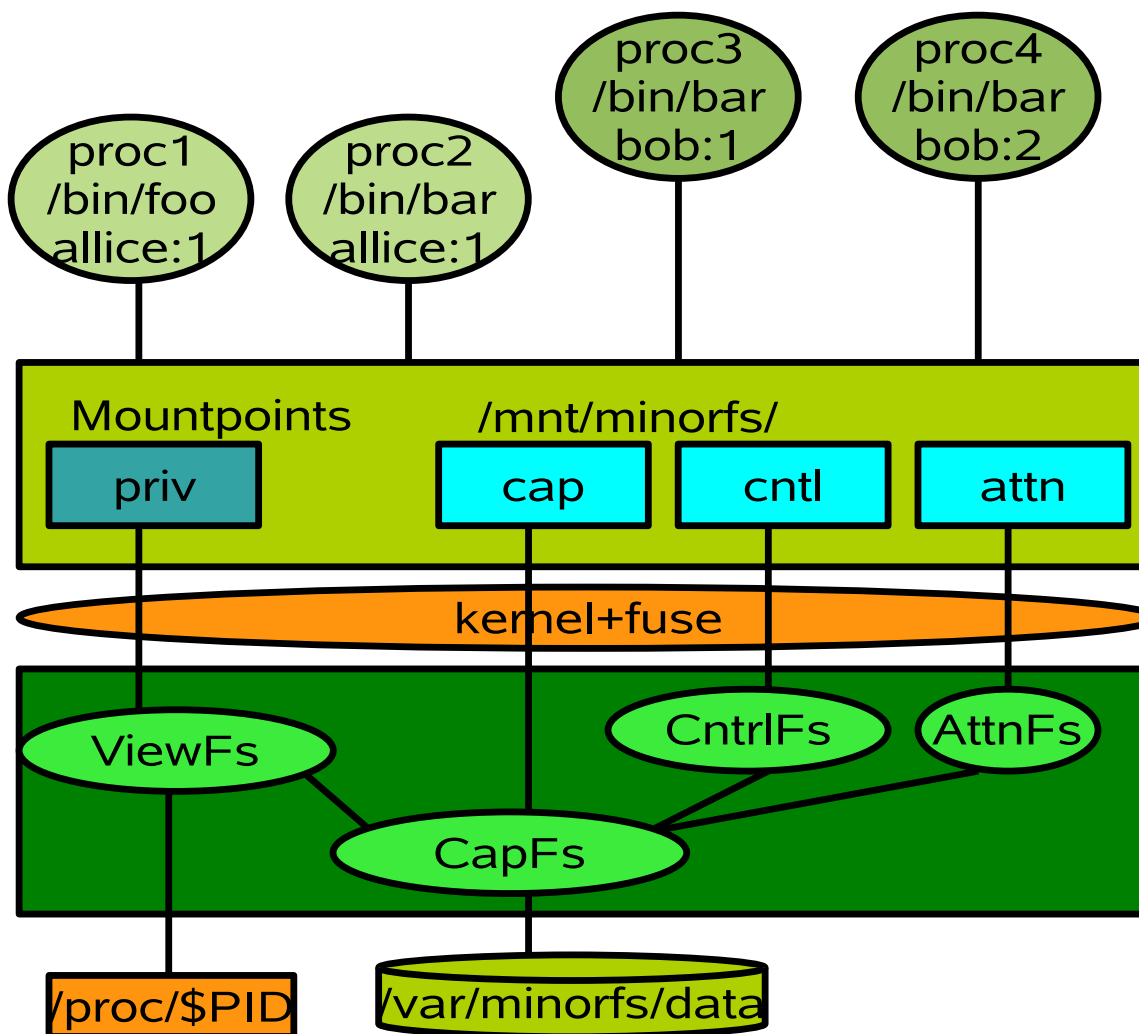
Personal and non commercial educational usage and
copying (as a whole) of this document for these purposes is permitted.
All further rights reserved.

View and capability based access control and delegation

This document tries to give a graphical representation of the outer workings of minorfs. MinorFs is a set of 4 simple cooperating userspace file systems that combine to create a practically usable (non performance optimized) implementation of POLP/POLA constructs as filesystem abstractions.

- Private data for (pseudo) persistent processes.
- Delegation
- Attenuation
- Revocation

Before we go into how MinorFs can be used to accomplish these 4, we will look at the basic system setup for a system running MinorFs



The normal processes can access MinorFs using 4 mountpoints under /mnt/minorfs. Each mount point is handled through the kernel and fuse by a userspace process responsible for one of four simple pseudo file systems. Of the four filesystem processes 3 run as 3 distinct unprivileged users. The 4th, *ViewFs*, needs to run as root in order to be allowed sufficient privileges to /proc/\$PID. AppArmor profiles are also supplied for each of the processes, as are uid based iptables rules.

CapFs & the storage tree.

The CapFs pseudo filesystem is a basic example of a minimal loop back file system implementation. It basically loops back the `/var/minorfs/data` directory structure to be made available through the `/mnt/minorfs/cap` mountpoint. It does this however with taking into account a few issues needed to allow the base level of least authority constructs.

All the files and directories under `/var/minorfs/data` are readable and writable only for the `capfs` user. All directories and files under `/mnt/minorfs/cap` are readable and writable to every user and process on the system. From the identity based access control models point of view you would say that there is no access control on `capfs`. There is however a twist to this.

If a process tries to list the content of `/mnt/minorfs/cap`, the directory listing will show an empty directory. Thus although all the files and directories under `/mnt/minorfs/cap` are accessible without restrictions, they are only accessible to processes that have knowledge of a valid path.

So you may now ask yourself: 'so a process can just guess a valid path under `/var/minorfs/data` and gain full access?'. The answer to this question is NO. Each file and directory under `/var.minorfs/data` is available from the `/mnt/minorfs/cap` mountpoint, but only when using a so called *strong path*.

Lets say we have a file:

```
/var/minorfs/data/userdata/1001/foo/bar.txt
```

CapFs will make this file available through 5 different paths. For example:

```
/mnt/minorfs/cap/b5a92cc7fdd2c7b8147a2596bf46af5def7edd13/userdata/1001/foo/bar.txt
```

```
/mnt/minorfs/cap/9aaab7f143ea9db2bc92321d598ebff07170e714/1001/foo/bar.txt
```

```
/mnt/minorfs/cap/c75498db49db4c7be7d3f3f9b56f71f5f553ba20/foo/bar.txt
```

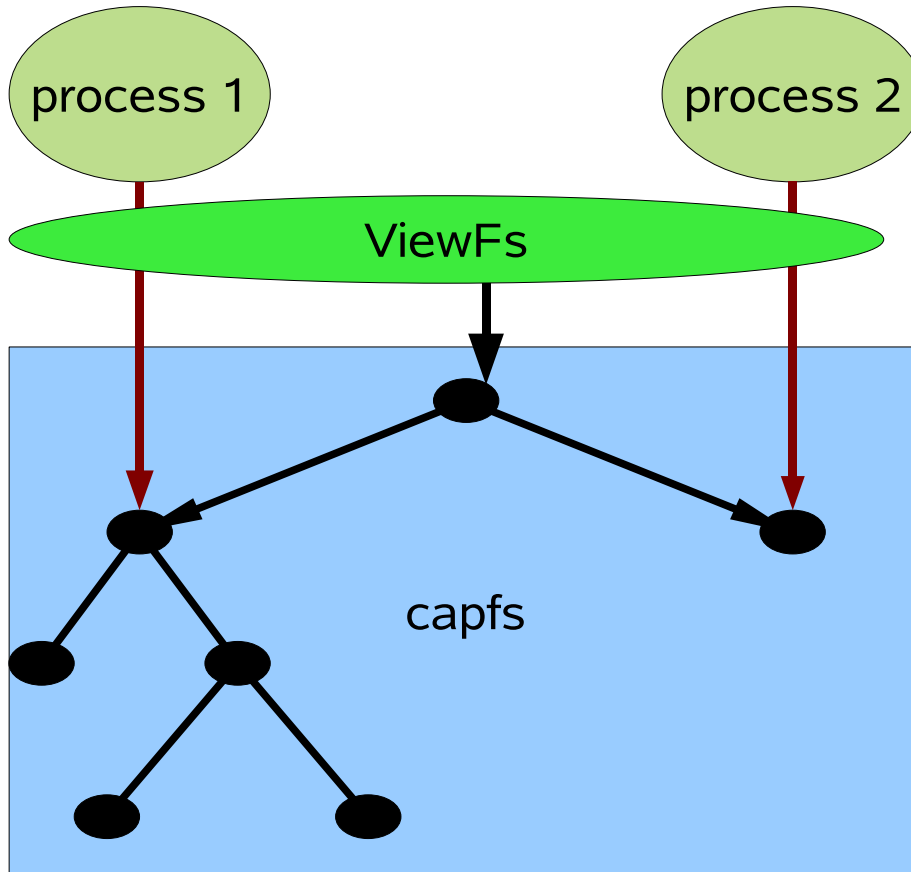
```
/mnt/minorfs/cap/4521f30385de93ec625b06c65c16697febf6549a/bar.txt
```

```
/mnt/minorfs/cap/f2486c3686df1772c4d2e6add8cd438d33340428
```

As you can see in the above example, each *node* in the `/var/minorfs/data` tree corresponds to a *strong path node* at the top level of the `/mnt/minorfs/cap/` file system. Thus every node in the `/var/minorfs/data` hierarchy is available without restrictions to any process with *knowledge* about the strong path of that node or one of its parent directory nodes.

If a process has a strong path token like `4521f30385de93ec625b06c65c16697febf6549a`, than this token can be used to designate the directory (`/var/minorfs/data/userdata/1001/foo` in this case), and to get full access to that node and any underlying node.

ViewFs and private directory structures.



When ViewFs is started it receives a strong path token into capfs that gives it access to the `/var/minorfs/data/user` directory. The task of MinorViewFs is to provide a private view to individual processes. It does this by first mapping the Linux non persistent process id's to persistent n-th-claim process id's. For each n-th claim persistent process id, ViewFs creates two private branches under `/var/minorfs/data/user/<uid>`. The strong paths for these two private branches are made available by ViewFs in the form of two symbolic links under `/mnt/minorfs/priv`.

ViewFs has access to `/proc/$PID` for each process accessing it. This allows ViewFs to use information about the process to map each program run by a uid to some unique identifier.

In an idealized version, N-th claim process id's would look like: *alice@/bin/foo:1* .

How this should be read is:

The process running under the user id of **alice** that is an instance of **/bin/foo** and has claimed the **1st** slot. The use of this scheme of n-th claim persistence should allow programs to truly implement persistent processes by storing all their relevant state under */mnt/minorfs/priv*.

Please note that each combination of user and executable has its own set of slots defined. Thus the processes shown on the first diagram could have n-th claim process id's like:

- *alice@/bin/foo:1*
- *alice@/bin/bar:1*
- *bob@/bin/bar:1*
- *bob@/bin/bar:2*

Each of these four processes would get its own private directories made available as distinct symbolic links under the */mnt/minorfs/priv* mountpoint.

It is important to note that many circumstances prohibit ViewFs from using this approach to persistent process identifiers. Things like behavior changing command line arguments and environment variables, dynamically loaded library modules, interpreted languages etc make that instead of */bin/foo*, ViewFs needs to use an identifying digest instead.

This means that a persistent process id would look something like:

- [*alice@255ee6c685cabe118db76e1fefdda7569e89294e:1*](#)

ViewFs maintains two symbolic links for each process.

- */mnt/minorfs/priv/home*
- */mnt/minorfs/priv/tmp*

The first symlink points to persistent storage for the current persistent process id. The second points to a private temporary directory that ViewFs shall clean up at the point in time that the process ends its execution. You could say that some legacy programs may want to use */mnt/minorfs/priv/home* in place of */home/\$USER* and */mnt/minorfs/priv/tmp* in place of */tmp*.

One useful example of such a construct for the home directory would be that of the secure shell.

The ssh program allows the use of key based access control. That is, the client has access to a private key under *\$HOME/.ssh*, while the server has access to the public key.

If a user does not put a pass phrase on his private key, a compromised web browser may take his private key and send it to a hostile entity.

While current management tools for MinorFs are lacking, it is possible to move the *\$HOME/.ssh* directory and the corresponding private key to the private home directory of the (first instance of) ssh run by this user. This would result in the user being able to run ssh without using a password and without having to be concerned about compromised web browsers stealing his private key.

ViewFs & minoradmin

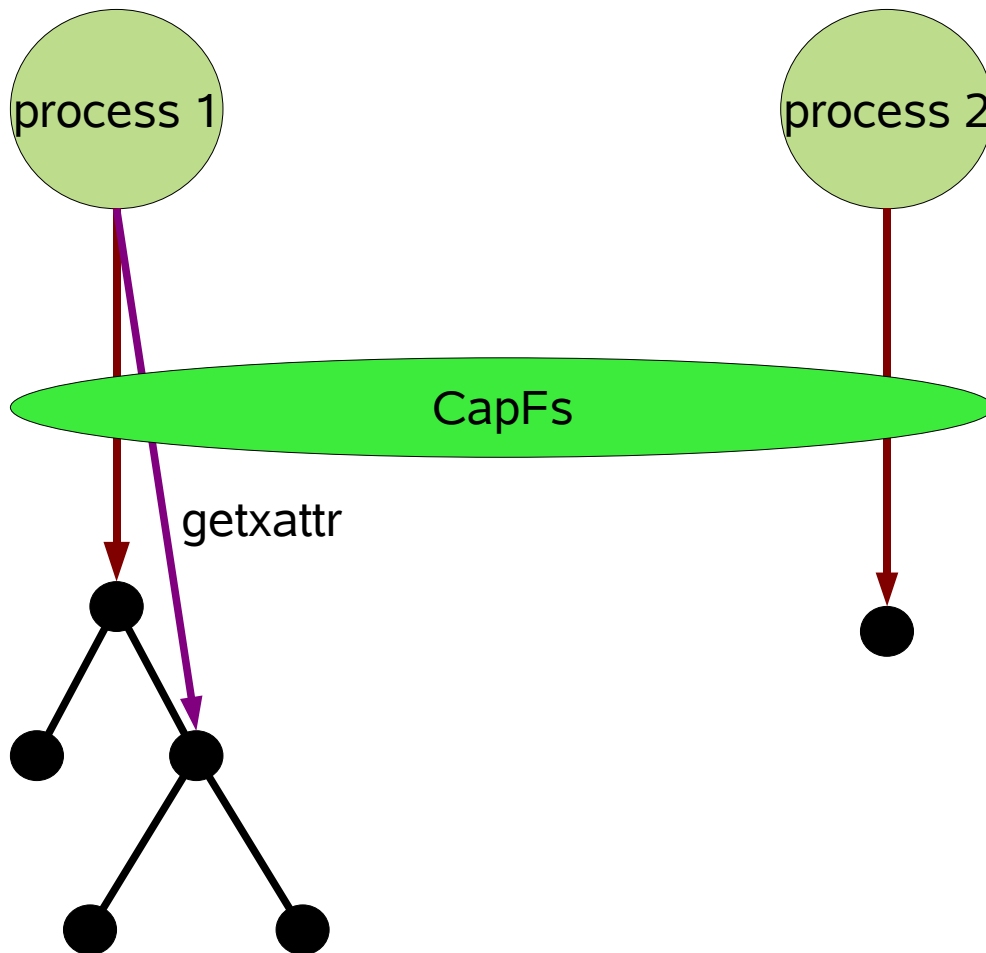
While ViewFs/Capfs combine to allow processes to keep their data safe from each other, doing so also shields the user from being able to directly access the data that a process chooses not to delegate. In most cases this *data hiding* is what allows the user to manage authority without the need to dump all manageable data into one big user global name space. There are however incidental instances where a user may legitimately need to (re)claim authority over all her data.

In order to accommodate this need, ViewFs gives special authority to the first instance for each user id of the /usr/local/bin/minoradmin program. ViewFs discloses access to the strong path that maps to /var/minorfs/data/\$UID.

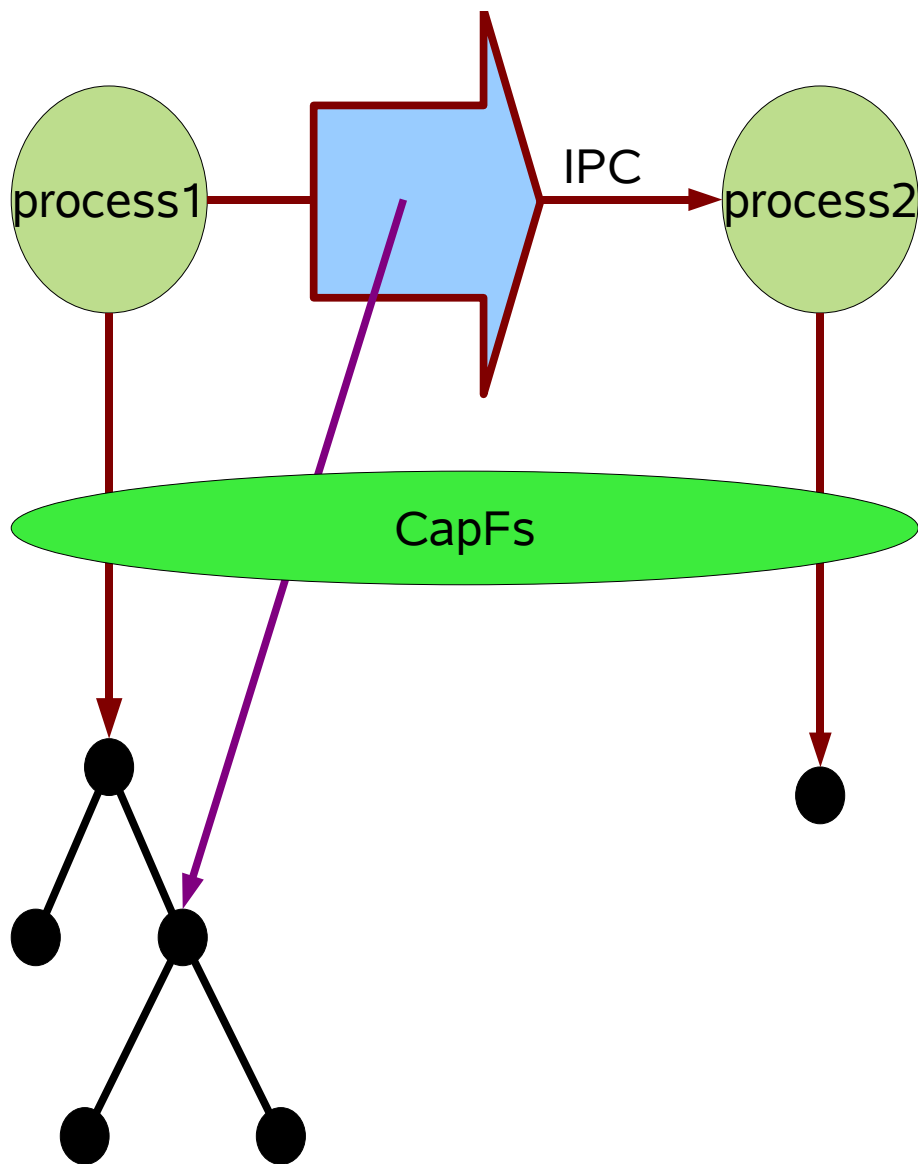
The minoradmin program is a trivially simple application. It maintains the digest of a password in a password file in its /mnt/minorfs/priv/home directory. On startup it asks the user for a password. If the password validates it releases its strong path to standard out so the user can use this to access all her data. If minoradmin does not find a passwd file in its home directory, it asks the user to create its authoritative password.

The minoradmin program allows the user *access* to all her authority in *exceptional* circumstances, without the need to require from the user to *manage* all her authority under *normal* circumstances. This in contrast to the usage of a powerbox in systems like Plash or Capdesk, where the role of the user and the application and the direction of delegation are such that the user has an abundance of authority she has to manage all of the time.

CapFs and delegation

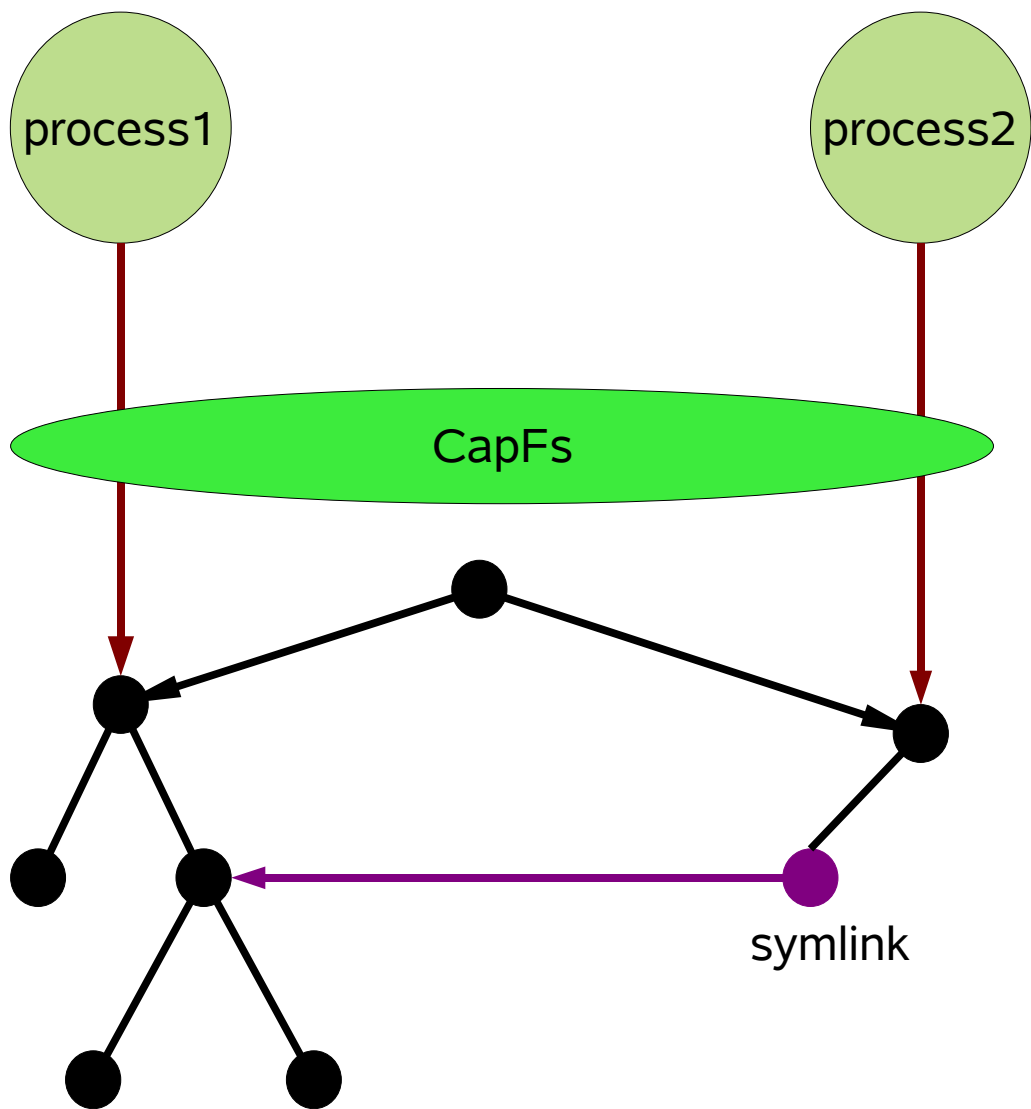


The way that CapFs allows the implementation of delegation is by use of the extended attributes for files and directories. MinorCapFs defines the extended attribute '*delegatable*'. Requesting the extended attribute *delegatable* from a file or directory, returns an alternative *strong* delegatable path to the same underlying file or directory. This alternative path consists of the prefix `/mnt/minorfs/cap/` followed by an unguessable file or directory name. You could consider the strong path token to be something equivalent to a password capability. Password capabilities are described in chapter 8 of this reader. You can find information on the usage of extended attributes in the `getxattr` and `listxattr` man pages in section 2 of your linux manuals.



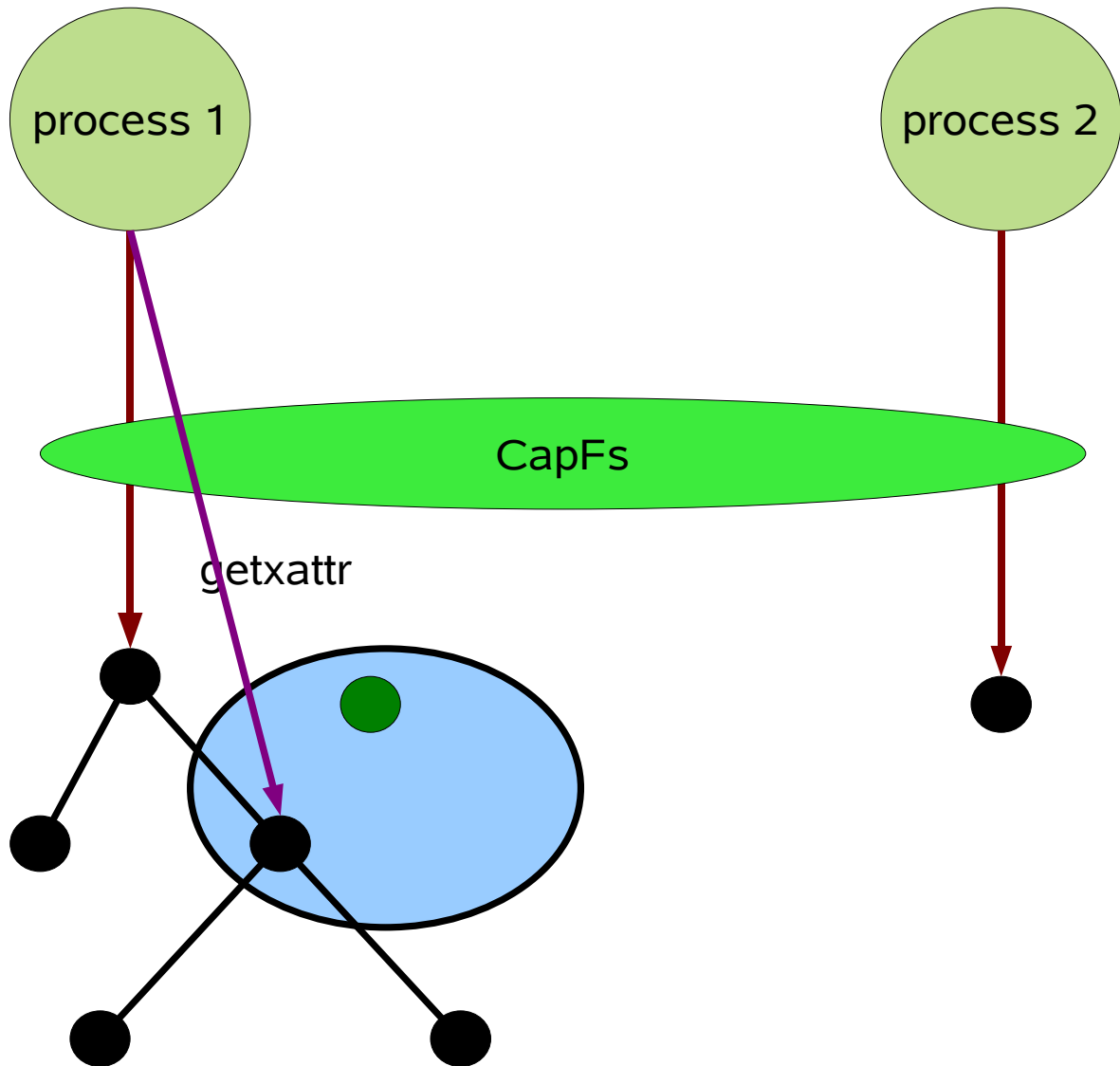
The path returned by CapfsFs extended attribute basically defines a view to a subbranch under `/var/minorfs/data` that is defined by the strong path. This strong path token or (password) capability is usable for each process that has knowledge of this unguessable path.

This means that any normal method of inter process communication can be used in order to delegate the (full) access to this sub branch. There are multiple forms of inter process communication available on the Linux platform, including Sys-V IPC, named pipes, unix domain sockets, TCP/IP sockets etc, most of which would be suitable for usage with MinorFs.



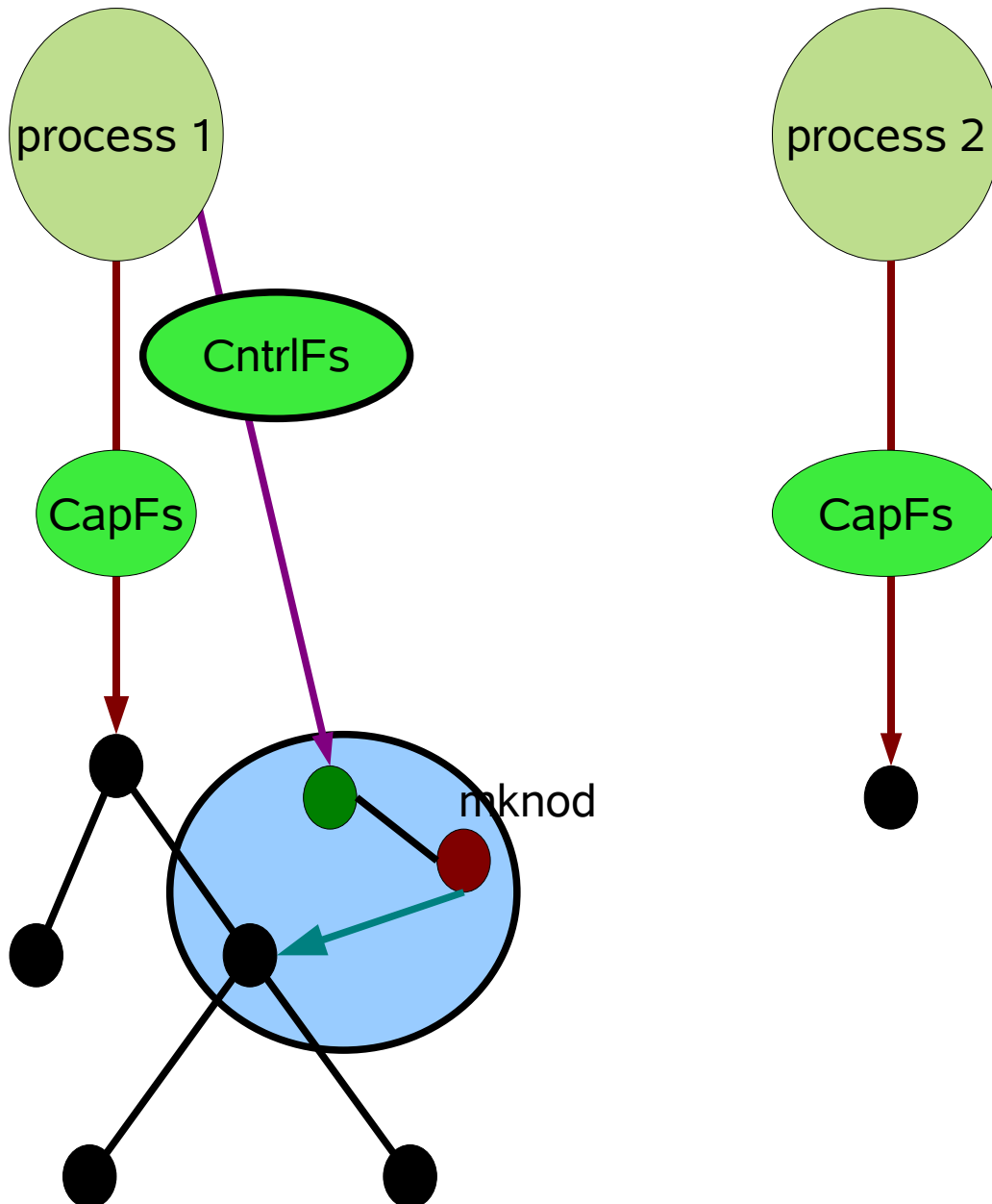
After a process receives a path into the CapFs hierarchy, it can in a very simple way integrate this path into its private view. The way it does this is by using the path to create a symbolic link. For information on the usage of symbolic links, consult the symlink manual in section 2 of your Linux manual pages..

CapFs and attenuation



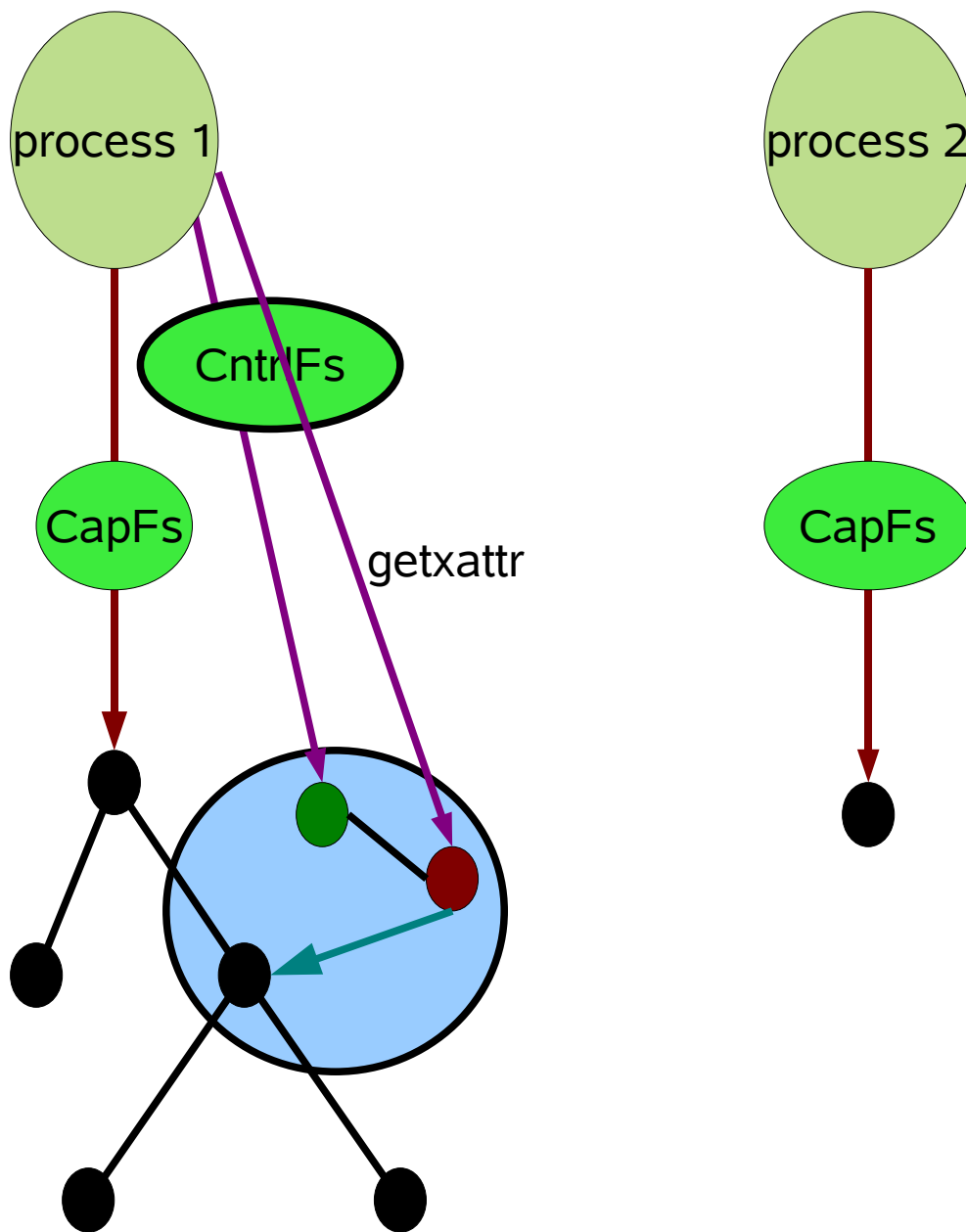
The same way that CapFs provides the possibility to retrieve an alternative delegatable path to a private file or directory, it provides a similar functionality to retrieve a path into the CntrlFs file system. In this CntrlFs file system each private node is represented as a directory of delegatable attenuations for that node.

CntrlFs



Until now the delegated subgraphs all carried with full authority. Given that we would like to approach *least* authority, we need to introduce more file system abstractions to accommodate attenuation patterns. The first file system abstraction we shall look at is CntrlFs. The CntrlFs path returned by CapFs points to a directory. This directory can be used to create special files that control individually controllable attenuations for the original private node and its children. The special files can either be created by opening and closing, or by using the *mknod* call (see the *mknod, open* and *close* manual in section 2 of your Linux manual). Where normally Linux file and directory nodes have 3 sets of read/write/executable bits, one for the owning uid (user id), one for the owning gid (group id) and one for all others, these concepts have no meaning in MinorFs given the fact that MinorFs uses a completely different access model. MinorFs does NOT support the executable bit. and instead of 'user/group/other', it uses the 3 sets of r/w bits to define access to 'node', 'child directory nodes' and 'child file nodes'.

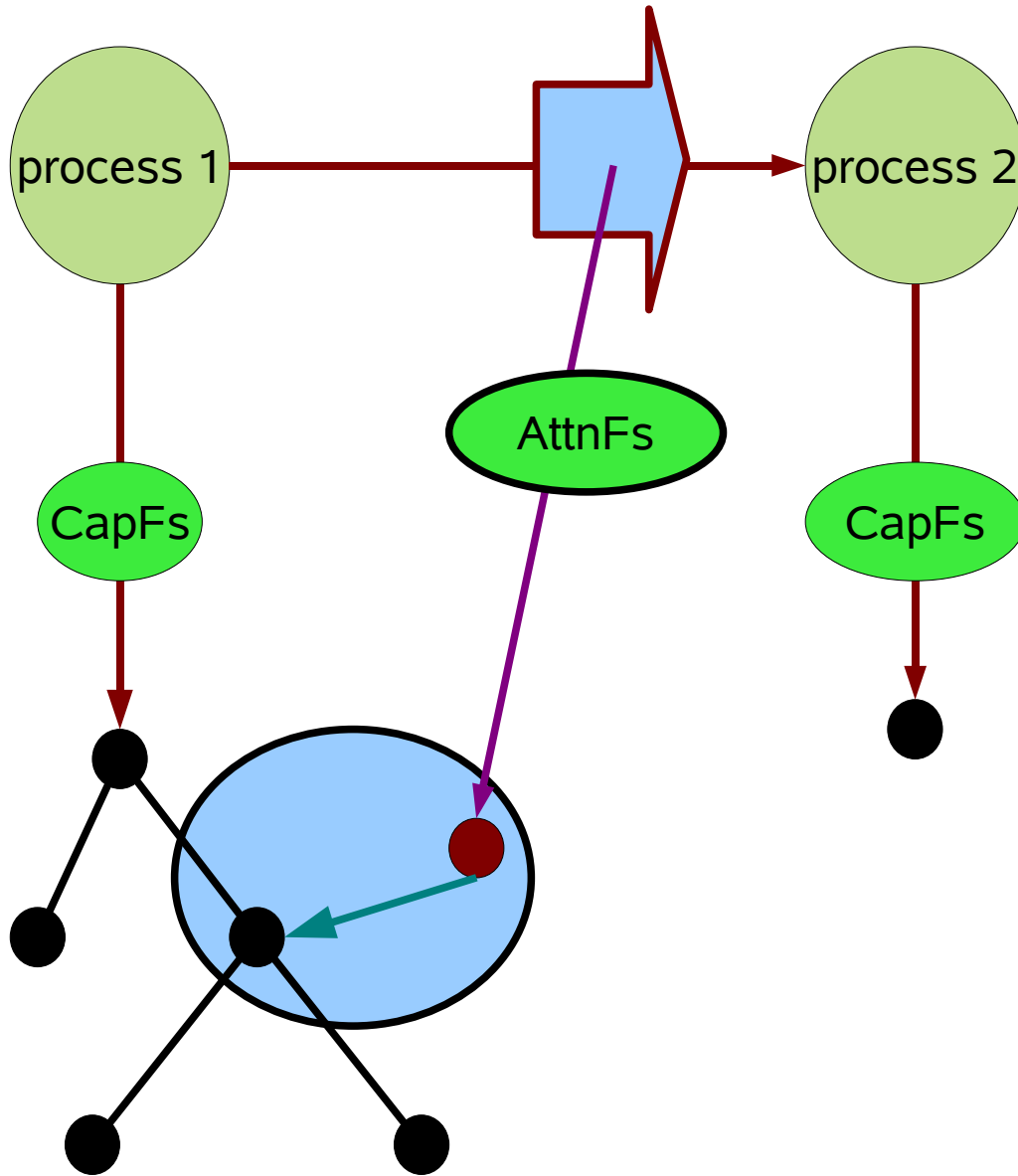
MinorControlFs and delegation



The same way that CapFs can give delegatable paths for its node's, CntrlFs can also provide paths to its node's. Other than with CapFs however, the paths provided by CntrlFs don't carry the same authority to the CntrlFs.

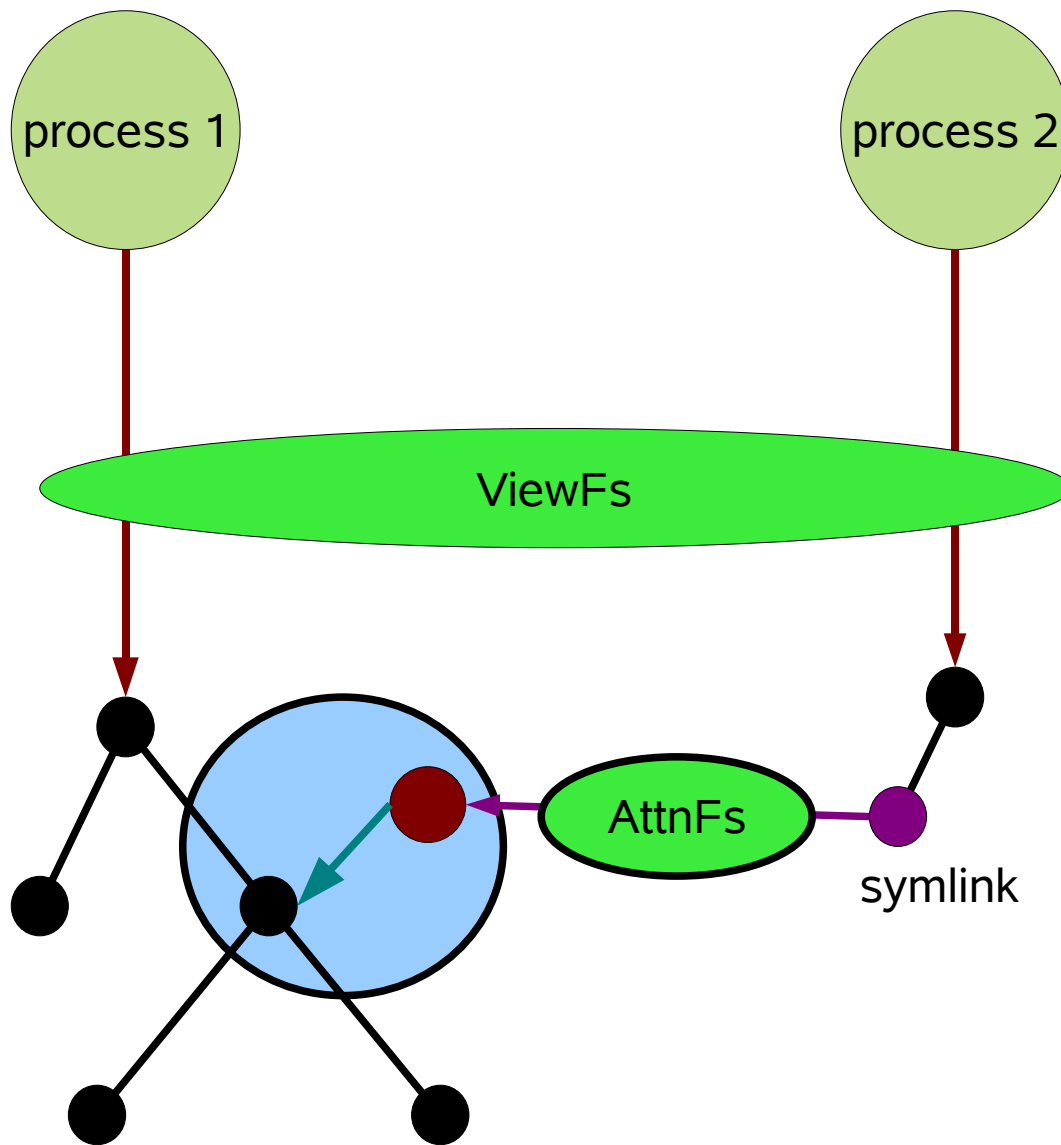
Instead the paths that CntrlFs returns for its control node's are paths into the AttnFs file system that are actually views on the original tree that have the attenuation defined by the control node enforced. This way, for example a read only password capability can be retrieved if the node in CntrlFs had its write bits set to zero.

AttnFs



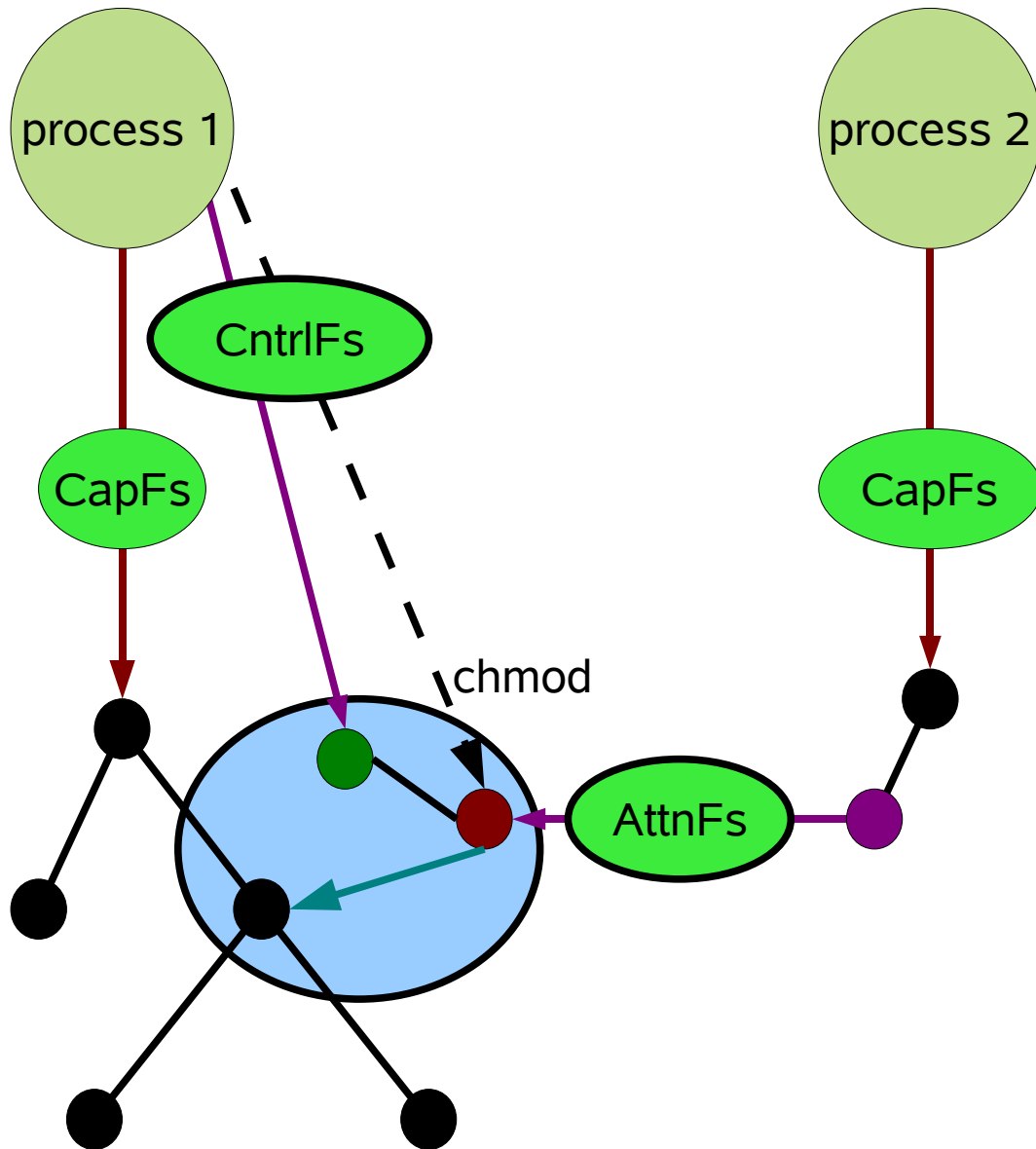
As with CapFs paths, AttnFs paths can be delegated using many forms of available inter process communication. This way a process can for example delegate read only access to part of its private directory tree.

MinorViewFs: completing attenuated delegation



Again CapFs is essential in completing delegation in a persistent way. In order to make the delegation persistent, CapFs can be used to create a symbolic link in the private directory tree of the receiving process that points to the attenuation node.

CntrlFs and revocation



In some special cases, a process that gave attenuated access to a resource may want to later revoke this access. CntrlFs makes such a thing possible by allowing the use of the chmod operation on existing control nodes. By this mechanism, the attenuation bits can be dynamically changed.

Future additional filesystem abstractions.

The above four file system abstractions combine to show the basis of dynamic discretionary access control mechanisms, and how they can be used to implement:

- Data Hiding / encapsulation
- Decomposition
- Composition
- Attenuation
- Revocation

There are however two essential constructs missing that are often useful for doing least authority:

- Copy On Write (COW).
- Transactional access.

Future versions of MinorFs may include additional file system abstractions that implement those.

Copy On Write allows a directory tree that is available read only to be made available as if it was a rw accessible directory tree, by providing alternative storage for any changes made to the original.

COW would basically be a composition pattern. The CapFs would need to provide an additional attribute for directory nodes, allowing the user to delegate a directory as 'raw' storage location for a COW filesystem abstraction. This attribute pointing into the CowCtrlFs filesystem and carrying authority to the directory in the CapFs filesystem could be used in a *symlink* operation to bind the raw storage directory to a read only (or used as read only) source directory, thus turning a read only directory into a COW version of that directory. The newly created CowCtrlFs node would then have an extended attribute that points to the base of a COW directory node in the CowFs filesystem. The CowFs filesystem should provide for the same decomposition and attenuation possibilities that CapFs does.

Transactional access allows one process to give access to a file or directory tree only for the duration of a single transaction. It allows the process to be sure that the authority granted to the other entity is not stored and/or accumulated to be used outside of the temporal scope of the transaction. Although the attenuation and revocation mechanisms provided by CntrlFs and AttnFs could be used to provide for transactional access controls, a simpler better fitted solution for this is desirable in order to truly address the temporal dimension of least authority in a way attractive to developers. The most programmer friendly representation of a transaction would be that of an open file handle. A TransactFs implementation should thus allow creating, binding and opening a transaction node that would yield unattenuated access to a target tree/file until the associated file handle is closed, at what point the strong path of the targeted node, and each of its underlying nodes will instantly be invalidated.

Although COW and transactional access are important constructs, implementation of these in MinorFs is to be delayed in order to address other issues. Please consult <http://polacanthus.net/> for more information on the current state and functionality of MinorFs.