NAME

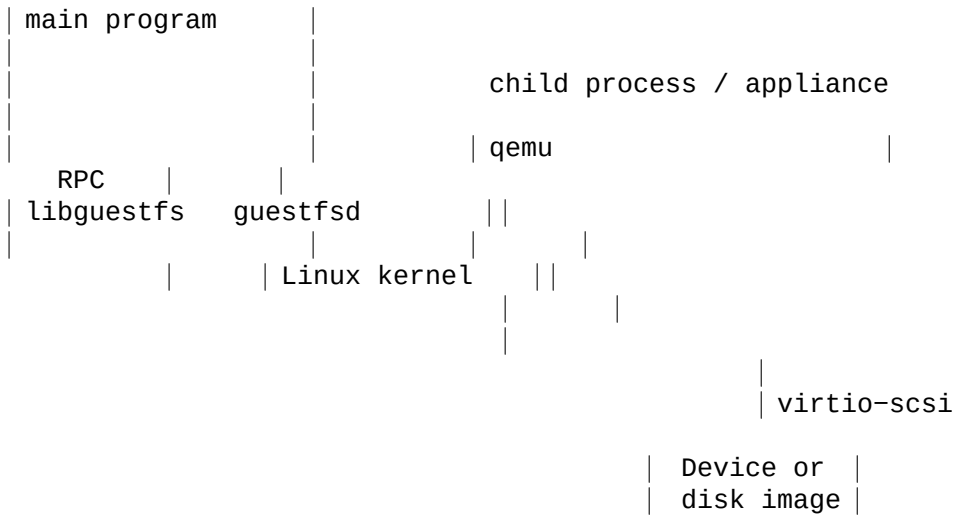
guestfs−internals − architecture and internals of libguestfs

DESCRIPTION

This manual page is for hackers who want to understand how libguestfs works internally. This is just a description of how libguestfs works now, and it may change at any time in the future.

ARCHITECTURE

Internally, libguestfs is implemented by running an appliance (a special type of small virtual machine) using **qemu** (1). Qemu runs as a child process of the main program.



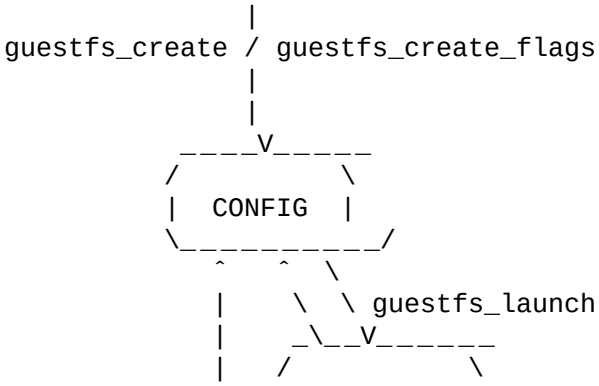
The library, linked to the main program, creates the child process and hence the appliance in the ‘‘guestfs_launch’’ in **guestfs** (3) function.

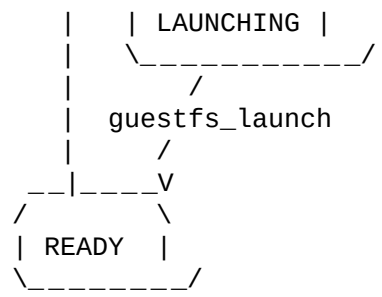
Inside the appliance is a Linux kernel and a complete stack of userspace tools (such as LVM and ext2 programs) and a small controlling daemon called ‘‘guestfsd’’. The library talks to ‘‘guestfsd’’ using remote procedure calls (RPC). There is a mostly one-to-one correspondence between libguestfs API calls and RPC calls to the daemon. Lastly the disk image(s) are attached to the qemu process which translates device access by the appliance’s Linux kernel into accesses to the image.

A common misunderstanding is that the appliance ‘‘is’’ the virtual machine. Although the disk image you are attached to might also be used by some virtual machine, libguestfs doesn’t know or care about this. (But you will care if both libguestfs’s qemu process and your virtual machine are trying to update the disk image at the same time, since these usually results in massive disk corruption).

STATE MACHINE

libguestfs uses a state machine to model the child process:





The normal transitions are (1) CONFIG (when the handle is created, but there is no child process), (2) LAUNCHING (when the child process is booting up), (3) READY meaning the appliance is up, actions can be issued to, and carried out by, the child process.

The guest may be killed by ‘‘guestfs_kill_subprocess’’ in **guestfs** (3), or may die asynchronously at any time (eg. due to some internal error), and that causes the state to transition back to CONFIG.

Configuration commands for qemu such as ‘‘guestfs_set_path’’ in **guestfs** (3) can only be issued when in the CONFIG state.

The API offers one call that goes from CONFIG through LAUNCHING to READY. ‘‘guestfs_launch’’ in **guestfs** (3) blocks until the child process is READY to accept commands (or until some failure or timeout). ‘‘guestfs_launch’’ in **guestfs** (3) internally moves the state from CONFIG to LAUNCHING while it is running.

API actions such as ‘‘guestfs_mount’’ in **guestfs** (3) can only be issued when in the READY state. These API calls block waiting for the command to be carried out. There are no non-blocking versions, and no way to issue more than one command per handle at the same time.

Finally, the child process sends asynchronous messages back to the main program, such as kernel log messages. You can register a callback to receive these messages.

INTERNALS

APPLIANCE BOOT PROCESS

This process has evolved and continues to evolve. The description here corresponds only to the current version of libguestfs and is provided for information only.

In order to follow the stages involved below, enable libguestfs debugging (set the environment variable `LIBGUESTFS_DEBUG=1`).

Create the appliance

`supermin --build` is invoked to create the kernel, a small initrd and the appliance.

The appliance is cached in */var/tmp/.guestfs-<UID>* (or in another directory if `LIBGUESTFS_CACHEDIR` or `TMPDIR` are set).

For a complete description of how the appliance is created and cached, read the **supermin** (1) man page.

Start qemu and boot the kernel

qemu is invoked to boot the kernel.

Run the initrd

`supermin --build` builds a small initrd. The initrd is not the appliance. The purpose of the initrd is to load enough kernel modules in order that the appliance itself can be mounted and started.

The initrd is a cpio archive called */var/tmp/.guestfs-<UID>/appliance.d/initrd*.

When the initrd has started you will see messages showing that kernel modules are being loaded, similar to this:

```
supermin: ext2 mini initrd starting up
supermin: mounting /sys
supermin: internal insmod libcrc32c.ko
supermin: internal insmod crc32c-intel.ko
```

Find and mount the appliance device

The appliance is a sparse file containing an ext2 filesystem which contains a familiar (although reduced in size) Linux operating system. It would normally be called */var/tmp/.guestfs-<UID>/appliance.d/root*.

The regular disks being inspected by libguestfs are the first devices exposed by qemu (eg. as */dev/vda*).

The last disk added to qemu is the appliance itself (eg. */dev/vdb* if there was only one regular disk).

Thus the final job of the initrd is to locate the appliance disk, mount it, and switch root into the appliance, and run */init* from the appliance.

If this works successfully you will see messages such as:

```
supermin: picked /sys/block/vdb/dev as root device
supermin: creating /dev/root as block special 252:16
supermin: mounting new root on /root
supermin: chroot
Starting /init script ...
```

Note that `Starting /init script ...` indicates that the appliance's init script is now running.

Initialize the appliance

The appliance itself now initializes itself. This involves starting certain processes like `udev`, possibly printing some debug information, and finally running the daemon (`guestfsd`).

The daemon

Finally the daemon (`guestfsd`) runs inside the appliance. If it runs you should see:

```
verbose daemon enabled
```

The daemon expects to see a named virtio-serial port exposed by qemu and connected on the other end to the library.

The daemon connects to this port (and hence to the library) and sends a four byte message `GUESTFS_LAUNCH_FLAG`, which initiates the communication protocol (see below).

COMMUNICATION PROTOCOL

Don't rely on using this protocol directly. This section documents how it currently works, but it may change at any time.

The protocol used to talk between the library and the daemon running inside the qemu virtual machine is a simple RPC mechanism built on top of XDR (RFC 1014, RFC 1832, RFC 4506).

The detailed format of structures is in *common/protocol/guestfs_protocol.x* (note: this file is automatically generated).

There are two broad cases, ordinary functions that don't have any `FileIn` and `FileOut` parameters, which are handled with very simple request/reply messages. Then there are functions that have any `FileIn` or `FileOut` parameters, which use the same request and reply messages, but they may also be followed by files sent using a chunked encoding.

ORDINARY FUNCTIONS (NO FILEIN/FILEOUT PARAMS)

For ordinary functions, the request message is:

```
total length (header + arguments,
     but not including the length word itself)
struct guestfs_message_header (encoded as XDR)
struct guestfs_<foo>_args (encoded as XDR)
```

The total length field allows the daemon to allocate a fixed size buffer into which it slurps the rest of the message. As a result, the total length is limited to`GUESTFS_MESSAGE_MAX` bytes (currently 4MB), which means the effective size of any request is limited to somewhere under this size.

Note also that many functions don't take any arguments, in which case the `guestfs_`*`foo`*`_args` is completely omitted.

The header contains the procedure number (`guestfs_proc`) which is how the receiver knows what type of args structure to expect, or none at all.

For functions that take optional arguments, the optional arguments are encoded in the `guestfs_`*`foo`*`_args` structure in the same way as ordinary arguments. A bitmask in the header indicates which optional arguments are meaningful. The bitmask is also checked to see if it contains bits set which the daemon does not know about (eg. if more optional arguments were added in a later version of the library), and this causes the call to be rejected.

The reply message for ordinary functions is:

```
total length (header + ret,
     but not including the length word itself)
struct guestfs_message_header (encoded as XDR)
struct guestfs_<foo>_ret (encoded as XDR)
```

As above the `guestfs_`*`foo`*`_ret` structure may be completely omitted for functions that return no formal return values.

As above the total length of the reply is limited to `GUESTFS_MESSAGE_MAX`.

In the case of an error, a flag is set in the header, and the reply message is slightly changed:

```
total length (header + error,
     but not including the length word itself)
struct guestfs_message_header (encoded as XDR)
struct guestfs_message_error (encoded as XDR)
```

The `guestfs_message_error` structure contains the error message as a string.

FUNCTIONS THAT HAVE FILEIN PARAMETERS

A `FileIn` parameter indicates that we transfer a file *into* the guest. The normal request message is sent (see above). However this is followed by a sequence of file chunks.

```
total length (header + arguments,
     but not including the length word itself,
     and not including the chunks)
struct guestfs_message_header (encoded as XDR)
struct guestfs_<foo>_args (encoded as XDR)
sequence of chunks for FileIn param #0
sequence of chunks for FileIn param #1 etc.
```

The ‘‘sequence of chunks’’ is:

```
length of chunk (not including length word itself)
struct guestfs_chunk (encoded as XDR)
length of chunk
struct guestfs_chunk (encoded as XDR)
  ...
length of chunk
struct guestfs_chunk (with data.data_len == 0)
```

The final chunk has the `data_len` field set to zero. Additionally a flag is set in the final chunk to indicate either successful completion or early cancellation.

At time of writing there are no functions that have more than one FileIn parameter. However this is (theoretically) supported, by sending the sequence of chunks for each FileIn parameter one after another (from left to right).

Both the library (sender) *and* the daemon (receiver) may cancel the transfer. The library does this by sending a chunk with a special flag set to indicate cancellation. When the daemon sees this, it cancels the whole RPC, does *not* send any reply, and goes back to reading the next request.

The daemon may also cancel. It does this by writing a special word `GUESTFS_CANCEL_FLAG` to the socket. The library listens for this during the transfer, and if it gets it, it will cancel the transfer (it sends a cancel chunk). The special word is chosen so that even if cancellation happens right at the end of the transfer (after the library has finished writing and has started listening for the reply), the ‘‘spurious’’ cancel flag will not be confused with the reply message.

This protocol allows the transfer of arbitrary sized files (no 32 bit limit), and also files where the size is not known in advance (eg. from pipes or sockets). However the chunks are rather small (`GUESTFS_MAX_CHUNK_SIZE`), so that neither the library nor the daemon need to keep much in memory.

FUNCTIONS THAT HAVE FILEOUT PARAMETERS

The protocol for FileOut parameters is exactly the same as for FileIn parameters, but with the roles of daemon and library reversed.

```
total length (header + ret,
     but not including the length word itself,
     and not including the chunks)
struct guestfs_message_header (encoded as XDR)
struct guestfs_<foo>_ret (encoded as XDR)
sequence of chunks for FileOut param #0
sequence of chunks for FileOut param #1 etc.
```

INITIAL MESSAGE

When the daemon launches it sends an initial word (`GUESTFS_LAUNCH_FLAG`) which indicates that the guest and daemon is alive. This is what ‘‘guestfs_launch’’ in **guestfs** (3) waits for.

PROGRESS NOTIFICATION MESSAGES

The daemon may send progress notification messages at any time. These are distinguished by the normal length word being replaced by `GUESTFS_PROGRESS_FLAG`, followed by a fixed size progress message.

The library turns them into progress callbacks (see ‘‘GUESTFS_EVENT_PROGRESS’’ in **guestfs** (3)) if there is a callback registered, or discards them if not.

The daemon self-limits the frequency of progress messages it sends (see `daemon/proto.c:notify_progress`). Not all calls generate progress messages.

FIXED APPLIANCE

When libguestfs (or libguestfs tools) are run, they search a path looking for an appliance. The path is built into libguestfs, or can be set using the `LIBGUESTFS_PATH` environment variable.

Normally a supermin appliance is located on this path (see ‘‘SUPERMIN APPLIANCE’’ in **supermin** (1)).

libguestfs reconstructs this into a full appliance by running `supermin --build`.

However, a simpler ‘‘fixed appliance’’ can also be used. libguestfs detects this by looking for a directory on the path containing all the following files:

- *kernel*
- *initrd*
- *root*
- *README.fixed* (note that it **must** be present as well)

If the fixed appliance is found, libguestfs skips supermin entirely and just runs the virtual machine (using qemu or the current backend, see ‘‘BACKEND’’ in **guestfs** (3)) with the kernel, initrd and root disk from the fixed appliance.

Thus the fixed appliance can be used when a platform or a Linux distribution does not support supermin. You build the fixed appliance on a platform that does support supermin using **libguestfs-make-fixed-appliance** (1), copy it over, and use that to run libguestfs.

SEE ALSO

guestfs (3), **guestfs-hacking** (1), **guestfs-examples** (3), **libguestfs-test-tool** (1), **libguestfs-make-fixed-appliance** (1), http://libguestfs.org/.

AUTHORS

Richard W.M. Jones (`rjones at redhat dot com`)

COPYRIGHT

LICENSE

BUGS

To get a list of bugs against libguestfs, use this link: https://bugzilla.redhat.com/buglist.cgi?component=libguestfs&product=Virtualization+Tools

To report a new bug against libguestfs, use this link: https://bugzilla.redhat.com/enter_bug.cgi?component=libguestfs&product=Virtualization+Tools

When reporting a bug, please supply:

- The version of libguestfs.
- Where you got libguestfs (eg. which Linux distro, compiled from source, etc)
- Describe the bug accurately and give a way to reproduce it.
- Run **libguestfs-test-tool** (1) and paste the **complete, unedited** output into the bug report.