NAME

Sys::Guestfs − Perl bindings for libguestfs

SYNOPSIS

```
 use Sys::Guestfs;

 my $g = Sys::Guestfs−>new ();
 $g−>add_drive_opts ('guest.img', format => 'raw');
 $g−>launch ();
 $g−>mount ('/dev/sda1', '/');
 $g−>touch ('/hello');
 $g−>shutdown ();
 $g−>close ();
```

DESCRIPTION

The `Sys::Guestfs` module provides a Perl XS binding to the libguestfs API for examining and modifying virtual machine disk images.

Amongst the things this is good for: making batch configuration changes to guests, getting disk used/free statistics (see also: virt-df), migrating between virtualization systems (see also: virt−p2v), performing partial backups, performing partial guest clones, cloning guests and changing registry/UUID/hostname info, and much else besides.

Libguestfs uses Linux kernel and qemu code, and can access any type of guest filesystem that Linux and qemu can, including but not limited to: ext2/3/4, btrfs, FAT and NTFS, LVM, many different disk partition schemes, qcow, qcow2, vmdk.

Libguestfs provides ways to enumerate guest storage (eg. partitions, LVs, what filesystem is in each LV, etc.). It can also run commands in the context of the guest. Also you can access filesystems over FUSE.

ERRORS

All errors turn into calls to `croak` (see **Carp** (3)).

The error string from libguestfs is directly available from `$@`. Use the `last_errno` method if you w ant to get the errno.

METHODS

`$g` = Sys::Guestfs−>new ([environment => 0,] [close_on_exit => 0]);

Create a new guestfs handle.

If the optional argument `environment` is false, then the `GUESTFS_CREATE_NO_ENVIRONMENT` flag is set.

If the optional argument `close_on_exit` is false, then the `GUESTFS_CREATE_NO_CLOSE_ON_EXIT` flag is set.

`$g`−>close ();

Explicitly close the guestfs handle.

Note: You should not usually call this function. The handle will be closed implicitly when its reference count goes to zero (eg. when it goes out of scope or the program ends). This call is only required in some exceptional cases, such as where the program may contain cached references to the handle 'somewhere' and you really have to have the close happen right away. After calling `close` the program must not call any method (including `close`) on the handle (but the implicit call to `DESTROY` that happens when the final reference is cleaned up is OK).

`$Sys::Guestfs::EVENT_CLOSE`

See "GUESTFS_EVENT_CLOSE" in **guestfs** (3).

`$Sys::Guestfs::EVENT_SUBPROCESS_QUIT`

See "GUESTFS_EVENT_SUBPROCESS_QUIT" in **guestfs** (3).

`$Sys::Guestfs::EVENT_LAUNCH_DONE`
: See "GUESTFS_EVENT_LAUNCH_DONE" in **guestfs**(3).

`$Sys::Guestfs::EVENT_PROGRESS`
: See "GUESTFS_EVENT_PROGRESS" in **guestfs**(3).

`$Sys::Guestfs::EVENT_APPLIANCE`
: See "GUESTFS_EVENT_APPLIANCE" in **guestfs**(3).

`$Sys::Guestfs::EVENT_LIBRARY`
: See "GUESTFS_EVENT_LIBRARY" in **guestfs**(3).

`$Sys::Guestfs::EVENT_TRACE`
: See "GUESTFS_EVENT_TRACE" in **guestfs**(3).

`$Sys::Guestfs::EVENT_ENTER`
: See "GUESTFS_EVENT_ENTER" in **guestfs**(3).

`$Sys::Guestfs::EVENT_LIBVIRT_AUTH`
: See "GUESTFS_EVENT_LIBVIRT_AUTH" in **guestfs**(3).

`$Sys::Guestfs::EVENT_WARNING`
: See "GUESTFS_EVENT_WARNING" in **guestfs**(3).

`$Sys::Guestfs::EVENT_ALL`
: See "GUESTFS_EVENT_ALL" in **guestfs**(3).

`$event_handle` = `$g`−>set_event_callback (\&cb, `$event_bitmask`);
: Register `cb` as a callback function for all of the events in `$event_bitmask` (one or more `$Sys::Guestfs::EVENT_*` flags logically or'd together).

  This function returns an event handle which can be used to delete the callback using `delete_event_callback`.

  The callback function receives 4 parameters:

```
 &cb ($event, $event_handle, $buf, $array)
```

  `$event`
  : The event which happened (equal to one of `$Sys::Guestfs::EVENT_*`).

  `$event_handle`
  : The event handle.

  `$buf`
  : For some event types, this is a message buffer (ie. a string).

  `$array`
  : For some event types (notably progress events), this is an array of integers.

  You should carefully read the documentation for "guestfs_set_event_callback" in **guestfs**(3) before using this function.

`$g`−>delete_event_callback ($event_handle);
: This removes the callback which was previously registered using `set_event_callback`.

`$str` = Sys::Guestfs::event_to_string ($events);
: `$events` is either a single event or a bitmask of events. This returns a printable string, useful for debugging.

  Note that this is a class function, not a method.

`$errnum` = `$g`−>last_errno ();
: This returns the last error number (errno) that happened on the handle `$g`.

  If successful, an errno integer not equal to zero is returned.

  If no error number is available, this returns 0. See "guestfs_last_errno" in **guestfs**(3) for more details of why this can happen.

You can use the standard Perl module **Errno** (3) to compare the numeric error returned from this call with symbolic errnos:

```
 $g->mkdir ("/foo");
 if ($g->last_errno() == Errno::EEXIST()) {
   # mkdir failed because the directory exists already.
 }
```

$g->acl_delete_def_file ($dir);
: This function deletes the default POSIX Access Control List (ACL) attached to directory `dir`.

  This function depends on the feature `acl`. See also`$g->feature-available`.

$acl = $g->acl_get_file ($path, $acltype);
: This function returns the POSIX Access Control List (ACL) attached to `path`. The ACL is returned in ‘‘long text form’’ (see **acl** (5)).

  The `acltype` parameter may be:

  `access`
  : Return the ordinary (access) ACL for any file, directory or other filesystem object.

  `default`
  : Return the default ACL. Normally this only makes sense if `path` is a directory.

  This function depends on the feature `acl`. See also`$g->feature-available`.

$g->acl_set_file ($path, $acltype, $acl);
: This function sets the POSIX Access Control List (ACL) attached to `path`.

  The `acltype` parameter may be:

  `access`
  : Set the ordinary (access) ACL for any file, directory or other filesystem object.

  `default`
  : Set the default ACL. Normally this only makes sense if `path` is a directory.

  The `acl` parameter is the new ACL in either ‘‘long text form’’ or ‘‘short text form’’ (see **acl** (5)). The new ACL completely replaces any previous ACL on the file. The ACL must contain the full Unix permissions (eg. `u::rwx,g::rx,o::rx`).

  If you are specifying individual users or groups, then the mask field is also required (eg. `m::rwx`), followed by the `u:`*`ID`*`:...` and/or `g:`*`ID`*`:...` field(s). A full ACL string might therefore look like this:

```
 u::rwx,g::rwx,o::rwx,m::rwx,u:500:rwx,g:500:rwx
 \ Unix permissions / \mask/ \      ACL        /
```

  You should use numeric UIDs and GIDs. To map usernames and groupnames to the correct numeric ID in the context of the guest, use the Augeas functions (see `$g->aug_init`).

  This function depends on the feature `acl`. See also`$g->feature-available`.

$g->add_cdrom ($filename);
: This function adds a virtual CD-ROM disk image to the guest.

  The image is added as read-only drive, so this function is equivalent of `$g->add_drive_ro`.

  This function is deprecated. In new code, use the ‘‘add_drive_ro’’ call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$nrdisks` = `$g`−>add_domain ($dom [, libvirturi => `$libvirturi`] [, readonly => `$readonly`] [, iface => `$iface`] [, live => `$live`] [, allowuuid => `$allowuuid`] [, readonlydisk => `$readonlydisk`] [, cachemode => `$cachemode`] [, discard => `$discard`] [, copyonread => `$copyonread`]);

This function adds the disk(s) attached to the named libvirt domain `dom`. It works by connecting to libvirt, requesting the domain and domain XML from libvirt, parsing it for disks, and calling `$g−>add_drive_opts` on each one.

The number of disks added is returned. This operation is atomic: if an error is returned, then no disks are added.

This function does some minimal checks to make sure the libvirt domain is not running (unless `readonly` is true). In a future version we will try to acquire the libvirt lock on each disk.

Disks must be accessible locally. This often means that adding disks from a remote libvirt connection (see <https://libvirt.org/remote.html>) will fail unless those disks are accessible via the same device path locally too.

The optional `libvirturi` parameter sets the libvirt URI (see <https://libvirt.org/uri.html>). If this is not set then we connect to the default libvirt URI (or one set through an environment variable, see the libvirt documentation for full details).

The optional `live` flag controls whether this call will try to connect to a running virtual machine `guestfsd` process if it sees a suitable <channel> element in the libvirt XML definition. The default (if the flag is omitted) is never to try. See ''ATTACHING TO RUNNING DAEMONS'' in **guestfs** (3) for more information.

If the `allowuuid` flag is true (default is false) then a UUID *may* be passed instead of the domain name. The`dom` string is treated as a UUID first and look ed up, and if that lookup fails then we treat `dom` as a name as usual.

The optional `readonlydisk` parameter controls what we do for disks which are marked <readonly/> in the libvirt XML. Possible values are:

readonlydisk = ''error''

If `readonly` is false:

The whole call is aborted with an error if any disk with the <readonly/> flag is found.

If `readonly` is true:

Disks with the <readonly/> flag are added read-only.

readonlydisk = ''read''

If `readonly` is false:

Disks with the <readonly/> flag are added read-only. Other disks are added read/write.

If `readonly` is true:

Disks with the <readonly/> flag are added read-only.

readonlydisk = ''write'' (default)

If `readonly` is false:

Disks with the <readonly/> flag are added read/write.

If `readonly` is true:

Disks with the <readonly/> flag are added read-only.

readonlydisk = ''ignore''

If `readonly` is true or false:

Disks with the <readonly/> flag are skipped.

If present, the value of `logical_block_size` attribute of <blockio/> tag in libvirt XML will be passed as `blocksize` parameter to `$g->add_drive_opts`.

The other optional parameters are passed directly through to `$g->add_drive_opts`.

`$g`−>add_drive ($filename [, readonly => `$readonly`] [, format => `$format`] [, iface => `$iface`] [, name => `$name`] [, label => `$label`] [, protocol => `$protocol`] [, server => `$server`] [, username => `$username`] [, secret => `$secret`] [, cachemode => `$cachemode`] [, discard => `$discard`] [, copyonread => `$copyonread`] [, blocksize => `$blocksize`]);

This function adds a disk image called *filename* to the handle. *filename* may be a regular host file or a host device.

When this function is called before `$g->launch` (the usual case) then the first time you call this function, the disk appears in the API as */dev/sda*, the second time as */dev/sdb*, and so on.

In libguestfs X 1.20 you can also call this function after launch (with some restrictions). This is called ''hotplugging''. When hotplugging, you must specify a `label` so that the ne w disk gets a predictable name. For more information see ''HOTPLUGGING'' in **guestfs** (3).

You don't necessarily need to be root when using libguestfs. However you obviously do need sufficient permissions to access the filename for whatever operations you want to perform (ie. read access if you just want to read the image or write access if you want to modify the image).

This call checks that *filename* exists.

filename may be the special string `"/dev/null"`. See ''NULL DISKS'' in **guestfs** (3).

The optional arguments are:

`readonly`

If true then the image is treated as read-only. Writes are still allowed, but they are stored in a temporary snapshot overlay which is discarded at the end. The disk that you add is not modified.

`format`

This forces the image format. If you omit this (or use `$g->add_drive` or `$g->add_drive_ro`) then the format is automatically detected. Possible formats include `raw` and `qcow2`.

Automatic detection of the format opens you up to a potential security hole when dealing with untrusted raw-format images. See CVE−2010−3851 and RHBZ#642934. Specifying the format closes this security hole.

`iface`

This rarely-used option lets you emulate the behaviour of the deprecated `$g->add_drive_with_if` call (q.v.)

`name`

The name the drive had in the original guest, e.g. */dev/sdb*. This is used as a hint to the guest inspection process if it is available.

`label`

Give the disk a label. The label should be a unique, short string using *only* ASCII characters `[a−zA−Z]`. As well as its usual name in the API (such as */dev/sda*), the drive will also be named */dev/disk/guestfs/label*.

See ''DISK LABELS'' in **guestfs** (3).

`protocol`

The optional protocol argument can be used to select an alternate source protocol.

See also: ''REMOTE STORAGE'' in **guestfs** (3).

`protocol = ''file''`
: *filename* is interpreted as a local file or device. This is the default if the optional protocol parameter is omitted.

`protocol = ''ftp''|''ftps''|''http''|''https''|''tftp''`
: Connect to a remote FTP, HTTP or TFTP server. The`server` parameter must also be supplied − see below.

  See also: "FTP, HTTP AND TFTP" in **guestfs** (3)

`protocol = ''gluster''`
: Connect to the GlusterFS server. The`server` parameter must also be supplied − see below.

  See also: "GLUSTER" in **guestfs** (3)

`protocol = ''iscsi''`
: Connect to the iSCSI server. The`server` parameter must also be supplied − see belo w. The `username` parameter may be supplied. See below. The`secret` parameter may be supplied. See below.

  See also: "ISCSI" in **guestfs** (3).

`protocol = ''nbd''`
: Connect to the Network Block Device server. The`server` parameter must also be supplied − see below.

  See also: "NETWORK BLOCK DEVICE" in **guestfs** (3).

`protocol = ''rbd''`
: Connect to the Ceph (librbd/RBD) server. The`server` parameter must also be supplied − see below. The`username` parameter may be supplied. See belo w. The`secret` parameter may be supplied. See below.

  See also: "CEPH" in **guestfs** (3).

`protocol = ''sheepdog''`
: Connect to the Sheepdog server. The`server` parameter may also be supplied − see belo w.

  See also: "SHEEPDOG" in **guestfs** (3).

`protocol = ''ssh''`
: Connect to the Secure Shell (ssh) server.

  The `server` parameter must be supplied. The `username` parameter may be supplied. See below.

  See also: "SSH" in **guestfs** (3).

`server`
: For protocols which require access to a remote server, this is a list of server(s).

```
 Protocol       Number of servers required
 --------       --------------------------
 file           List must be empty or param not used at all
 ftp|ftps|http|https|tftp  Exactly one
 gluster        Exactly one
 iscsi          Exactly one
 nbd            Exactly one
 rbd            Zero or more
 sheepdog       Zero or more
 ssh            Exactly one
```

Each list element is a string specifying a server. The string must be in one of the following formats:

```
hostname
hostname:port
tcp:hostname
tcp:hostname:port
unix:/path/to/socket
```

If the port number is omitted, then the standard port number for the protocol is used (see */etc/services*).

`username`
For the `ftp`, `ftps`, `http`, `https`, `iscsi`, `rbd`, `ssh` and `tftp` protocols, this specifies the remote username.

If not given, then the local username is used for `ssh`, and no authentication is attempted for ceph. But note this sometimes may give unexpected results, for example if using the libvirt backend and if the libvirt backend is configured to start the qemu appliance as a special user such as `qemu.qemu`. If in doubt, specify the remote username you want.

`secret`
For the `rbd` protocol only, this specifies the XsecretX to use when connecting to the remote device. It must be base64 encoded.

If not given, then a secret matching the given username will be looked up in the default keychain locations, or if no username is given, then no authentication will be used.

`cachemode`
Choose whether or not libguestfs will obey sync operations (safe but slow) or not (unsafe but fast). The possible values for this string are:

`cachemode = ''writeback''`
This is the default.

Write operations in the API do not return until a **write** (2) call has completed in the host [but note this does not imply that anything gets written to disk].

Sync operations in the API, including implicit syncs caused by filesystem journalling, will not return until an **fdatasync** (2) call has completed in the host, indicating that data has been committed to disk.

`cachemode = ''unsafe''`
In this mode, there are no guarantees. Libguestfs may cache anything and ignore sync requests. This is suitable only for scratch or temporary disks.

`discard`
Enable or disable discard (a.k.a. trim or unmap) support on this drive. If enabled, operations such as `$g->fstrim` will be able to discard / make thin / punch holes in the underlying host file or device.

Possible discard settings are:

`discard = ''disable''`
Disable discard support. This is the default.

`discard = ''enable''`
Enable discard support. Fail if discard is not possible.

`discard = ''besteffort''`
Enable discard support if possible, but don't fail if it is not supported.

Since not all backends and not all underlying systems support discard, this is a good choice if you want to use discard if possible, but don't mind if it doesn't work.

`copyonread`
: The boolean parameter `copyonread` enables copy-on-read support. This only affects disk formats which have backing files, and causes reads to be stored in the overlay layer, speeding up multiple reads of the same area of disk.

  The default is false.

`blocksize`
: This parameter sets the sector size of the disk. Possible values are `512` (the default if the parameter is omitted) or `4096`. Use`4096` when handling an ' 'Advanced Format'' disk that uses 4K sector size (<https://en.wikipedia.org/wiki/Advanced_Format>).

  Only a subset of the backends support this parameter (currently only the libvirt and direct backends do).

`$g`−>add_drive_opts ($filename [, readonly => `$readonly`] [, format => `$format`] [, iface => `$iface`] [, name => `$name`] [, label => `$label`] [, protocol => `$protocol`] [, server => `$server`] [, username => `$username`] [, secret => `$secret`] [, cachemode => `$cachemode`] [, discard => `$discard`] [, copyonread => `$copyonread`] [, blocksize => `$blocksize`]);
: This is an alias of ''add_drive''.

`$g`−>add_drive_ro ($filename);
: This function is the equivalent of calling `$g->add_drive_opts` with the optional parameter `GUESTFS_ADD_DRIVE_OPTS_READONLY` set to 1, so the disk is added read-only, with the format being detected automatically.

`$g`−>add_drive_ro_with_if ($filename, `$iface`);
: This is the same as `$g->add_drive_ro` but it allows you to specify the QEMU interface emulation to use at run time.

  This function is deprecated. In new code, use the ''add_drive'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>add_drive_scratch ($size [, name => `$name`] [, label => `$label`] [, blocksize => `$blocksize`]);
: This command adds a temporary scratch drive to the handle. The `size` parameter is the virtual size (in bytes). The scratch drive is blank initially (all reads return zeroes until you start writing to it). The drive is deleted when the handle is closed.

  The optional arguments `name`, `label` and `blocksize` are passed through to `$g->add_drive_opts`.

`$g`−>add_drive_with_if ($filename, `$iface`);
: This is the same as `$g->add_drive` but it allows you to specify the QEMU interface emulation to use at run time.

  This function is deprecated. In new code, use the ''add_drive'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$nrdisks` = `$g`−>add_libvirt_dom ($dom [, readonly => `$readonly`] [, iface => `$iface`] [, live => `$live`] [, readonlydisk => `$readonlydisk`] [, cachemode => `$cachemode`] [, discard => `$discard`] [, copyonread => `$copyonread`]);
: This function adds the disk(s) attached to the libvirt domain `dom`. It works by requesting the domain XML from libvirt, parsing it for disks, and calling `$g->add_drive_opts` on each one.

  In the C API we declare `void *dom`, but really it has type `virDomainPtr dom`. This is so we don't need <libvirt.h>.

  The number of disks added is returned. This operation is atomic: if an error is returned, then no disks are added.

This function does some minimal checks to make sure the libvirt domain is not running (unless `readonly` is true). In a future version we will try to acquire the libvirt lock on each disk.

Disks must be accessible locally. This often means that adding disks from a remote libvirt connection (see <https://libvirt.org/remote.html>) will fail unless those disks are accessible via the same device path locally too.

The optional `live` flag controls whether this call will try to connect to a running virtual machine `guestfsd` process if it sees a suitable <channel> element in the libvirt XML definition. The default (if the flag is omitted) is never to try. See "ATTACHING TO RUNNING DAEMONS" in **guestfs** (3) for more information.

The optional `readonlydisk` parameter controls what we do for disks which are marked <readonly/> in the libvirt XML. See `$g->add_domain` for possible v alues.

If present, the value of `logical_block_size` attribute of <blockio/> tag in libvirt XML will be passed as `blocksize` parameter to `$g->add_drive_opts`.

The other optional parameters are passed directly through to `$g->add_drive_opts`.

`$g`−>aug_clear ($augpath);
: Set the value associated with `path` to `NULL`. This is the same as the **augtool** (1) `clear` command.

`$g`−>aug_close ();
: Close the current Augeas handle and free up any resources used by it. After calling this, you have to call `$g->aug_init` again before you can use any other Augeas functions.

`%nrnodescreated` = `$g`−>aug_defnode ($name, `$expr`, `$val`);
: Defines a variable `name` whose value is the result of evaluating `expr`.

If `expr` evaluates to an empty nodeset, a node is created, equivalent to calling `$g->aug_set expr`, `val`. `name` will be the nodeset containing that single node.

On success this returns a pair containing the number of nodes in the nodeset, and a boolean flag if a node was created.

`$nrnodes` = `$g`−>aug_defvar ($name, `$expr`);
: Defines an Augeas variable `name` whose value is the result of evaluating `expr`. If `expr` is NULL, then `name` is undefined.

On success this returns the number of nodes in `expr`, or `0` if `expr` evaluates to something which is not a nodeset.

`$val` = `$g`−>aug_get ($augpath);
: Look up the value associated with `path`. If `path` matches e xactly one node, the `value` is returned.

`$g`−>aug_init ($root, `$flags`);
: Create a new Augeas handle for editing configuration files. If there was any previous Augeas handle associated with this guestfs session, then it is closed.

You must call this before using any other `$g->aug_*` commands.

`root` is the filesystem root. `root` must not be NULL, use / instead.

The flags are the same as the flags defined in <augeas.h>, the logical *or* of the following integers:

`AUG_SAVE_BACKUP` = 1
: Keep the original file with a `.augsave` extension.

`AUG_SAVE_NEWFILE` = 2
: Save changes into a file with extension `.augnew`, and do not overwrite original. Overrides `AUG_SAVE_BACKUP`.

`AUG_TYPE_CHECK` = 4
Typecheck lenses.

This option is only useful when debugging Augeas lenses. Use of this option may require additional memory for the libguestfs appliance. You may need to set the `LIBGUESTFS_MEMSIZE` environment variable or call `$g->set_memsize`.

`AUG_NO_STDINC` = 8
Do not use standard load path for modules.

`AUG_SAVE_NOOP` = 16
Make save a no-op, just record what would have been changed.

`AUG_NO_LOAD` = 32
Do not load the tree in `$g->aug_init`.

To close the handle, you can call `$g->aug_close`.

To find out more about Augeas, see <http://augeas.net/>.

`$g`−>aug_insert ($augpath, `$label`, `$before`);
Create a new sibling `label` for `path`, inserting it into the tree before or after `path` (depending on the boolean flag `before`).

`path` must match exactly one existing node in the tree, and `label` must be a label, ie. not contain /, `*` or end with a bracketed index `[N]`.

`$label` = `$g`−>aug_label ($augpath);
The label (name of the last element) of the Augeas path expression `augpath` is returned. `augpath` must match exactly one node, else this function returns an error.

`$g`−>aug_load ();
Load files into the tree.

See `aug_load` in the Augeas documentation for the full gory details.

`@matches` = `$g`−>aug_ls ($augpath);
This is just a shortcut for listing `$g->aug_match path/*` and sorting the resulting nodes into alphabetical order.

`@matches` = `$g`−>aug_match ($augpath);
Returns a list of paths which match the path expression `path`. The returned paths are sufficiently qualified so that they match exactly one node in the current tree.

`$g`−>aug_mv ($src, `$dest`);
Move the node `src` to `dest`. `src` must match exactly one node. `dest` is overwritten if it exists.

`$nrnodes` = `$g`−>aug_rm ($augpath);
Remove `path` and all of its children.

On success this returns the number of entries which were removed.

`$g`−>aug_save ();
This writes all pending changes to disk.

The flags which were passed to `$g->aug_init` affect exactly how files are saved.

`$g`−>aug_set ($augpath, `$val`);
Set the value associated with `augpath` to `val`.

In the Augeas API, it is possible to clear a node by setting the value to NULL. Due to an oversight in the libguestfs API you cannot do that with this call. Instead you must use the `$g->aug_clear` call.

`$nodes` = `$g`−>aug_setm ($base, `$sub`, `$val`);
Change multiple Augeas nodes in a single operation. `base` is an e xpression matching multiple nodes. `sub` is a path expression relative to `base`. All nodes matching`base` are found, and then for each

node, `sub` is changed to `val`. `sub` may also be `NULL` in which case the `base` nodes are modified.

This returns the number of nodes modified.

`$g`−>aug_transform ($lens, `$file` [, remove => `$remove`]);

Add an Augeas transformation for the specified `lens` so it can handle `file`.

If `remove` is true (`false` by default), then the transformation is removed.

`$g`−>available (\@groups);

This command is used to check the availability of some groups of functionality in the appliance, which not all builds of the libguestfs appliance will be able to provide.

The libguestfs groups, and the functions that those groups correspond to, are listed in "AVAILABILITY" in **guestfs**(3). You can also fetch this list at runtime by calling `$g->available_all_groups`.

The argument `groups` is a list of group names, eg: `["inotify", "augeas"]` would check for the availability of the Linux inotify functions and Augeas (configuration file editing) functions.

The command returns no error if *all* requested groups are available.

It fails with an error if one or more of the requested groups is unavailable in the appliance.

If an unknown group name is included in the list of groups then an error is always returned.

Notes:

- `$g->feature_available` is the same as this call, but with a slightly simpler to use API: that call returns a boolean true/false instead of throwing an error.
- You must call `$g->launch` before calling this function.

  The reason is because we don't know what groups are supported by the appliance/daemon until it is running and can be queried.
- If a group of functions is available, this does not necessarily mean that they will work. You still have to check for errors when calling individual API functions even if they are available.
- It is usually the job of distro packagers to build complete functionality into the libguestfs appliance. Upstream libguestfs, if built from source with all requirements satisfied, will support everything.
- This call was added in version `1.0.80`. In previous versions of libguestfs all you could do would be to speculatively execute a command to find out if the daemon implemented it. See also `$g->version`.

See also `$g->filesystem_available`.

`@groups` = `$g`−>available_all_groups ();

This command returns a list of all optional groups that this daemon knows about. Note this returns both supported and unsupported groups. To find out which ones the daemon can actually support you have to call `$g->available` / `$g->feature_available` on each member of the returned list.

See also `$g->available`, `$g->feature_available` and "AVAILABILITY" in **guestfs**(3).

`$g`−>base64_in ($base64file, `$filename`);

This command uploads base64−encoded data from `base64file` to *filename*.

`$g`−>base64_out ($filename, `$base64file`);

This command downloads the contents of *filename*, writing it out to local file `base64file` encoded as base64.

`$g`−>blkdiscard ($device);

This discards all blocks on the block device `device`, giving the free space back to the host.

This operation requires support in libguestfs, the host filesystem, qemu and the host kernel. If this support isn't present it may give an error or even appear to run but do nothing. You must also set the `discard` attribute on the underlying drive (see `$g->add_drive_opts`).

This function depends on the feature `blkdiscard`. See also`$g->feature-available`.

`$zeroes` = `$g`−>blkdiscardzeroes ($device);

This call returns true if blocks on `device` that have been discarded by a call to `$g->blkdiscard` are returned as blocks of zero bytes when read the next time.

If it returns false, then it may be that discarded blocks are read as stale or random data.

This function depends on the feature `blkdiscardzeroes`. See also `$g->feature-available`.

`%info` = `$g`−>blkid ($device);

This command returns block device attributes for `device`. The following fields are usually present in the returned hash. Other fields may also be present.

`UUID`
The uuid of this device.

`LABEL`
The label of this device.

`VERSION`
The version of blkid command.

`TYPE`
The filesystem type or RAID of this device.

`USAGE`
The usage of this device, for example `filesystem` or `raid`.

`$g`−>blockdev_flushbufs ($device);

This tells the kernel to flush internal buffers associated with `device`.

This uses the **blockdev** (8) command.

`$blocksize` = `$g`−>blockdev_getbsz ($device);

This returns the block size of a device.

Note: this is different from both *size in blocks* and *filesystem block size*. Also this setting is not really used by anything. You should probably not use it for anything. Filesystems have their own idea about what block size to choose.

This uses the **blockdev** (8) command.

`$ro` = `$g`−>blockdev_getro ($device);

Returns a boolean indicating if the block device is read-only (true if read-only, false if not).

This uses the **blockdev** (8) command.

`$sizeinbytes` = `$g`−>blockdev_getsize64 ($device);

This returns the size of the device in bytes.

See also `$g->blockdev_getsz`.

This uses the **blockdev** (8) command.

`$sectorsize` = `$g`−>blockdev_getss ($device);

This returns the size of sectors on a block device. Usually 512, but can be larger for modern devices.

(Note, this is not the size in sectors, use `$g->blockdev_getsz` for that).

This uses the **blockdev** (8) command.

`$sizeinsectors` = `$g`−>blockdev_getsz ($device);

This returns the size of the device in units of 512−byte sectors (even if the sectorsize isn't 512 bytes ... weird).

See also `$g->blockdev_getss` for the real sector size of the device, and `$g->blockdev_getsize64` for the more useful *size in bytes*.

This uses the **blockdev** (8) command.

`$g`−>blockdev_rereadpt ($device);

Reread the partition table on `device`.

This uses the **blockdev** (8) command.

`$g`−>blockdev_setbsz ($device, `$blocksize`);

This call does nothing and has never done anything because of a bug in blockdev. **Do not use it.**

If you need to set the filesystem block size, use the `blocksize` option of `$g->mkfs`.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs** (3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>blockdev_setra ($device, `$sectors`);

Set readahead (in 512−byte sectors) for the device.

This uses the **blockdev** (8) command.

`$g`−>blockdev_setro ($device);

Sets the block device named `device` to read-only.

This uses the **blockdev** (8) command.

`$g`−>blockdev_setrw ($device);

Sets the block device named `device` to read-write.

This uses the **blockdev** (8) command.

`$g`−>btrfs_balance_cancel ($path);

Cancel a running balance on a btrfs filesystem.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

`$g`−>btrfs_balance_pause ($path);

Pause a running balance on a btrfs filesystem.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

`$g`−>btrfs_balance_resume ($path);

Resume a paused balance on a btrfs filesystem.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

`%status` = `$g`−>btrfs_balance_status ($path);

Show the status of a running or paused balance on a btrfs filesystem.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

`$g`−>btrfs_device_add (\@devices, `$fs`);

Add the list of device(s) in `devices` to the btrfs filesystem mounted at `fs`. If`devices` is an empty list, this does nothing.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

$g−>btrfs_device_delete (\@devices, $fs);
Remove the devices from the btrfs filesystem mounted at fs. If devices is an empty list, this does nothing.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_filesystem_balance ($fs);
Balance the chunks in the btrfs filesystem mounted at fs across the underlying devices.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_filesystem_defragment ($path [, flush => $flush] [, compress => $compress]);
Defragment a file or directory on a btrfs filesystem. compress is one of zlib or lzo.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_filesystem_resize ($mountpoint [, size => $size]);
This command resizes a btrfs filesystem.

Note that unlike other resize calls, the filesystem has to be mounted and the parameter is the mountpoint not the device (this is a requirement of btrfs itself).

The optional parameters are:

size
The new size (in bytes) of the filesystem. If omitted, the filesystem is resized to the maximum size.

See also **btrfs** (8).

This function depends on the feature btrfs. See also $g->feature−available.

@devices = $g−>btrfs_filesystem_show ($device);
Show all the devices where the filesystems in device is spanned over.

If not all the devices for the filesystems are present, then this function fails and the errno is set to ENODEV.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_filesystem_sync ($fs);
Force sync on the btrfs filesystem mounted at fs.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_fsck ($device [, superblock => $superblock] [, repair => $repair]);
Used to check a btrfs filesystem, device is the device file where the filesystem is stored.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_image (\@source, $image [, compresslevel => $compresslevel]);
This is used to create an image of a btrfs filesystem. All data will be zeroed, but metadata and the like is preserved.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_qgroup_assign ($src, $dst, $path);
Add qgroup src to parent qgroup dst. This command can group several qgroups into a parent qgroup to share common limit.

This function depends on the feature btrfs. See also $g->feature−available.

$g−>btrfs_qgroup_create ($qgroupid, $subvolume);
Create a quota group (qgroup) for subvolume at subvolume.

This function depends on the feature btrfs. See also $g->feature−available.

$g->btrfs_qgroup_destroy ($qgroupid, $subvolume);
: Destroy a quota group.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_qgroup_limit ($subvolume, $size);
: Limit the size of the subvolume with path `subvolume`.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_qgroup_remove ($src, $dst, $path);
: Remove qgroup `src` from the parent qgroup `dst`.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

@qgroups = $g->btrfs_qgroup_show ($path);
: Show all subvolume quota groups in a btrfs filesystem, including their usages.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_quota_enable ($fs, $enable);
: Enable or disable subvolume quota support for filesystem which contains `path`.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_quota_rescan ($fs);
: Trash all qgroup numbers and scan the metadata again with the current config.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_replace ($srcdev, $targetdev, $mntpoint);
: Replace device of a btrfs filesystem. On a live filesystem, duplicate the data to the target device which is currently stored on the source device. After completion of the operation, the source device is wiped out and removed from the filesystem.

  The `targetdev` needs to be same size or larger than the `srcdev`. Devices which are currently mounted are never allowed to be used as the `targetdev`.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_rescue_chunk_recover ($device);
: Recover the chunk tree of btrfs filesystem by scanning the devices one by one.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_rescue_super_recover ($device);
: Recover bad superblocks from good copies.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_scrub_cancel ($path);
: Cancel a running scrub on a btrfs filesystem.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_scrub_resume ($path);
: Resume a previously canceled or interrupted scrub on a btrfs filesystem.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

$g->btrfs_scrub_start ($path);
: Reads all the data and metadata on the filesystem, and uses checksums and the duplicate copies from RAID storage to identify and repair any corrupt data.

  This function depends on the feature `btrfs`. See also`$g->feature-available`.

`%status` = `$g`−>btrfs_scrub_status ($path);
: Show status of running or finished scrub on a btrfs filesystem.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_set_seeding ($device, `$seeding`);
: Enable or disable the seeding feature of a device that contains a btrfs filesystem.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_subvolume_create ($dest [, qgroupid => `$qgroupid`]);
: Create a btrfs subvolume. The `dest` ar gument is the destination directory and the name of the subvolume, in the form */path/to/dest/name*. The optional parameter `qgroupid` represents the qgroup which the newly created subvolume will be added to.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_subvolume_create_opts ($dest [, qgroupid => `$qgroupid`]);
: This is an alias of ''btrfs_subvolume_create''.

`$g`−>btrfs_subvolume_delete ($subvolume);
: Delete the named btrfs subvolume or snapshot.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$id` = `$g`−>btrfs_subvolume_get_default ($fs);
: Get the default subvolume or snapshot of a filesystem mounted at `mountpoint`.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`@subvolumes` = `$g`−>btrfs_subvolume_list ($fs);
: List the btrfs snapshots and subvolumes of the btrfs filesystem which is mounted at `fs`.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_subvolume_set_default ($id, `$fs`);
: Set the subvolume of the btrfs filesystem `fs` which will be mounted by default. See `$g->btrfs_subvolume_list` to get a list of subvolumes.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`%btrfssubvolumeinfo` = `$g`−>btrfs_subvolume_show ($subvolume);
: Return detailed information of the subvolume.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_subvolume_snapshot ($source, `$dest` [, ro => `$ro`] [, qgroupid => `$qgroupid`]);
: Create a snapshot of the btrfs subvolume `source`. The `dest` ar gument is the destination directory and the name of the snapshot, in the form */path/to/dest/name*. By default the newly created snapshot is writable, if the value of optional parameter `ro` is true, then a readonly snapshot is created. The optional parameter `qgroupid` represents the qgroup which the newly created snapshot will be added to.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfs_subvolume_snapshot_opts ($source, `$dest` [, ro => `$ro`] [, qgroupid => `$qgroupid`]);
: This is an alias of ''btrfs_subvolume_snapshot''.

`$g`−>btrfstune_enable_extended_inode_refs ($device);
: This will Enable extended inode refs.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfstune_enable_skinny_metadata_extent_refs ($device);
: This enable skinny metadata extent refs.

  This function depends on the feature `btrfs`. See also `$g->feature-available`.

`$g`−>btrfstune_seeding ($device, `$seeding`);

Enable seeding of a btrfs device, this will force a fs readonly so that you can use it to build other filesystems.

This function depends on the feature `btrfs`. See also`$g−>feature−available`.

`$ptr` = `$g`−>c_pointer ();

In non-C language bindings, this allows you to retrieve the underlying C pointer to the handle (ie. `$g−>h *`). The purpose of this is to allow other libraries to interwork with libguestfs.

`$canonical` = `$g`−>canonical_device_name ($device);

This utility function is useful when displaying device names to the user. It takes a number of irregular device names and returns them in a consistent format:

/dev/hdX
/dev/vdX

These are returned as */dev/sdX*. Note this works for device names and partition names. This is approximately the reverse of the algorithm described in "BLOCK DEVICE NAMING" in **guestfs** (3).

/dev/mapper/VG−LV
/dev/dm−N

Converted to */dev/VG/LV* form using `$g−>lvm_canonical_lv_name`.

Other strings are returned unmodified.

`$cap` = `$g`−>cap_get_file ($path);

This function returns the Linux capabilities attached to `path`. The capabilities set is returned in text form (see **cap_to_text** (3)).

If no capabilities are attached to a file, an empty string is returned.

This function depends on the feature `linuxcaps`. See also`$g−>feature−available`.

`$g`−>cap_set_file ($path, `$cap`);

This function sets the Linux capabilities attached to `path`. The capabilities set`cap` should be passed in text form (see **cap_from_text** (3)).

This function depends on the feature `linuxcaps`. See also`$g−>feature−available`.

`$rpath` = `$g`−>case_sensitive_path ($path);

This can be used to resolve case insensitive paths on a filesystem which is case sensitive. The use case is to resolve paths which you have read from Windows configuration files or the Windows Registry, to the true path.

The command handles a peculiarity of the Linux ntfs−3g filesystem driver (and probably others), which is that although the underlying filesystem is case-insensitive, the driver exports the filesystem to Linux as case-sensitive.

One consequence of this is that special directories such as *C:\windows* may appear as */WINDOWS* or */windows* (or other things) depending on the precise details of how they were created. In Windows itself this would not be a problem.

Bug or feature? You decide: <https://www.tuxera.com/community/ntfs−3g−faq/#posixfilenames1>

`$g−>case_sensitive_path` attempts to resolve the true case of each element in the path. It will return a resolved path if either the full path or its parent directory exists. If the parent directory exists but the full path does not, the case of the parent directory will be correctly resolved, and the remainder appended unmodified. For example, if the file `"/Windows/System32/netkvm.sys"` exists:

`$g−>case_sensitive_path` ("/windows/system32/netkvm.sys")

"Windows/System32/netkvm.sys"

`$g->case_sensitive_path` ("/windows/system32/NoSuchFile")
"Windows/System32/NoSuchFile"

`$g->case_sensitive_path` ("/windows/system33/netkvm.sys")
ERROR

Note: Because of the above behaviour, `$g->case_sensitive_path` cannot be used to check for the existence of a file.

Note: This function does not handle drive names, backslashes etc.

See also `$g->realpath`.

`$content` = `$g`->cat ($path);

Return the contents of the file named `path`.

Because, in C, this function returns a `char *`, there is no way to differentiate between a `\0` character in a file and end of string. To handle binary files, use the `$g->read_file` or `$g->download` functions.

`$checksum` = `$g`->checksum ($csumtype, `$path`);

This call computes the MD5, SHAx or CRC checksum of the file named `path`.

The type of checksum to compute is given by the `csumtype` parameter which must have one of the following values:

`crc`
Compute the cyclic redundancy check (CRC) specified by POSIX for the `cksum` command.

`md5`
Compute the MD5 hash (using the **md5sum** (1) program).

`sha1`
Compute the SHA1 hash (using the **sha1sum** (1) program).

`sha224`
Compute the SHA224 hash (using the **sha224sum** (1) program).

`sha256`
Compute the SHA256 hash (using the **sha256sum** (1) program).

`sha384`
Compute the SHA384 hash (using the **sha384sum** (1) program).

`sha512`
Compute the SHA512 hash (using the **sha512sum** (1) program).

The checksum is returned as a printable string.

To get the checksum for a device, use `$g->checksum_device`.

To get the checksums for many files, use `$g->checksums_out`.

`$checksum` = `$g`->checksum_device ($csumtype, `$device`);

This call computes the MD5, SHAx or CRC checksum of the contents of the device named `device`. For the types of checksums supported see the `$g->checksum` command.

`$g`->checksums_out ($csumtype, `$directory`, `$sumsfile`);

This command computes the checksums of all regular files in *directory* and then emits a list of those checksums to the local output file `sumsfile`.

This can be used for verifying the integrity of a virtual machine. However to be properly secure you should pay attention to the output of the checksum command (it uses the ones from GNU coreutils). In particular when the filename is not printable, coreutils uses a special backslash syntax. For more information, see the GNU coreutils info file.

`$g`−>chmod ($mode, `$path`);

Change the mode (permissions) of `path` to `mode`. Only numeric modes are supported.

Note: When using this command from guestfish, `mode` by default would be decimal, unless you prefix it with `0` to get octal, ie. use `0700` not `700`.

The mode actually set is affected by the umask.

`$g`−>chown ($owner, `$group`, `$path`);

Change the file owner to `owner` and group to `group`.

Only numeric uid and gid are supported. If you want to use names, you will need to locate and parse the password file yourself (Augeas support makes this relatively easy).

`$count` = `$g`−>clear_backend_setting ($name);

If there is a backend setting string matching `"name"` or beginning with `"name="`, then that string is removed from the backend settings.

This call returns the number of strings which were removed (which may be 0, 1 or greater than 1).

See ''BACKEND'' in **guestfs** (3), ''BACKEND SETTINGS'' in **guestfs** (3).

`$output` = `$g`−>command (\@arguments);

This call runs a command from the guest filesystem. The filesystem must be mounted, and must contain a compatible operating system (ie. something Linux, with the same or compatible processor architecture).

The single parameter is an argv-style list of arguments. The first element is the name of the program to run. Subsequent elements are parameters. The list must be non-empty (ie. must contain a program name). Note that the command runs directly, and is *not* invoked via the shell (see `$g->sh`).

The return value is anything printed to *stdout* by the command.

If the command returns a non-zero exit status, then this function returns an error message. The error message string is the content of *stderr* from the command.

The `$PATH` environment variable will contain at least */usr/bin* and */bin*. If you require a program from another location, you should provide the full path in the first parameter.

Shared libraries and data files required by the program must be available on filesystems which are mounted in the correct places. It is the callerXs responsibility to ensure all filesystems that are needed are mounted at the right locations.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

`@lines` = `$g`−>command_lines (\@arguments);

This is the same as `$g->command`, but splits the result into a list of lines.

See also: `$g->sh_lines`

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

`$g`−>compress_device_out ($ctype, `$device`, `$zdevice` [, level => `$level`]);

This command compresses `device` and writes it out to the local file `zdevice`.

The `ctype` and optional `level` parameters have the same meaning as in `$g->compress_out`.

`$g`−>compress_out ($ctype, `$file`, `$zfile` [, level => `$level`]);

This command compresses *file* and writes it out to the local file *zfile*.

The compression program used is controlled by the `ctype` parameter. Currently this includes: `compress`, `gzip`, `bzip2`, `xz` or `lzop`. Some compression types may not be supported by particular builds of libguestfs, in which case you will get an error containing the substring ''not supported''.

The optional `level` parameter controls compression level. The meaning and default for this parameter depends on the compression program being used.

`$g`−>config ($hvparam, `$hvvalue`);

This can be used to add arbitrary hypervisor parameters of the form *−param value*. Actually itXs not quite arbitrary − we prevent you from setting some parameters which would interfere with parameters that we use.

The first character of `hvparam` string must be a `−` (dash).

`hvvalue` can be NULL.

`$g`−>copy_attributes ($src, `$dest` [, all => `$all`] [, mode => `$mode`] [, xattributes => `$xattributes`] [, ownership => `$ownership`]);

Copy the attributes of a path (which can be a file or a directory) to another path.

By default **no** attribute is copied, so make sure to specify any (or `all` to copy everything).

The optional arguments specify which attributes can be copied:

`mode`
: Copy part of the file mode from `source` to `destination`. Only the UNIX permissions and the sticky/setuid/setgid bits can be copied.

`xattributes`
: Copy the Linux extended attributes (xattrs) from `source` to `destination`. This flag does nothing if the *linuxxattrs* feature is not available (see `$g−>feature_available`).

`ownership`
: Copy the owner uid and the group gid of `source` to `destination`.

`all`
: Copy **all** the attributes from `source` to `destination`. Enabling it enables all the other flags, if they are not specified already.

`$g`−>copy_device_to_device ($src, `$dest` [, srcoffset => `$srcoffset`] [, destoffset => `$destoffset`] [, size => `$size`] [, sparse => `$sparse`] [, append => `$append`]);

The four calls `$g−>copy_device_to_device`, `$g−>copy_device_to_file`, `$g−>copy_file_to_device`, and `$g−>copy_file_to_file` let you copy from a source (device|file) to a destination (device|file).

Partial copies can be made since you can specify optionally the source offset, destination offset and size to copy. These values are all specified in bytes. If not given, the offsets both default to zero, and the size defaults to copying as much as possible until we hit the end of the source.

The source and destination may be the same object. However overlapping regions may not be copied correctly.

If the destination is a file, it is created if required. If the destination file is not large enough, it is extended.

If the destination is a file and the `append` flag is not set, then the destination file is truncated. If the `append` flag is set, then the copy appends to the destination file. The `append` flag currently cannot be set for devices.

If the `sparse` flag is true then the call avoids writing blocks that contain only zeroes, which can help in some situations where the backing disk is thin-provisioned. Note that unless the target is already zeroed, using this option will result in incorrect copying.

`$g`−>copy_device_to_file ($src, `$dest` [, srcoffset => `$srcoffset`] [, destoffset => `$destoffset`] [, size => `$size`] [, sparse => `$sparse`] [, append => `$append`]);

See `$g−>copy_device_to_device` for a general overview of this call.

$g−>copy_file_to_device ($src, $dest [, srcoffset => $srcoffset] [, destoffset => $destoffset] [, size => $size] [, sparse => $sparse] [, append => $append]);

  See $g−>copy_device_to_device for a general overview of this call.

$g−>copy_file_to_file ($src, $dest [, srcoffset => $srcoffset] [, destoffset => $destoffset] [, size => $size] [, sparse => $sparse] [, append => $append]);

  See $g−>copy_device_to_device for a general overview of this call.

  This is **not** the function you want for copying files. This is for copying blocks within existing files. See $g−>cp, $g−>cp_a and $g−>mv for general file copying and moving functions.

$g−>copy_in ($localpath, $remotedir);

  $g−>copy_in copies local files or directories recursively into the disk image, placing them in the directory called remotedir (which must exist).

  Wildcards cannot be used.

$g−>copy_out ($remotepath, $localdir);

  $g−>copy_out copies remote files or directories recursively out of the disk image, placing them on the host disk in a local directory called localdir (which must exist).

  To download to the current directory, use . as in:

```
 C<$g−E<gt>copy_out> /home .
```

  Wildcards cannot be used.

$g−>copy_size ($src, $dest, $size);

  This command copies exactly size bytes from one source device or file src to another destination device or file dest.

  Note this will fail if the source is too short or if the destination is not large enough.

  This function is deprecated. In new code, use the ‘‘copy_device_to_device’’ call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>cp ($src, $dest);

  This copies a file from src to dest where dest is either a destination filename or destination directory.

$g−>cp_a ($src, $dest);

  This copies a file or directory from src to dest recursively using the cp −a command.

$g−>cp_r ($src, $dest);

  This copies a file or directory from src to dest recursively using the cp −rP command.

  Most users should use $g−>cp_a instead. This command is useful when you don't want to preserve permissions, because the target filesystem does not support it (primarily when writing to DOS FAT filesystems).

$g−>cpio_out ($directory, $cpiofile [, format => $format]);

  This command packs the contents of *directory* and downloads it to local file cpiofile.

  The optional format parameter can be used to select the format. Only the following formats are currently permitted:

  newc

    New (SVR4) portable format. This format happens to be compatible with the cpio-like format used by the Linux kernel for initramfs.

    This is the default format.

`crc`
New (SVR4) portable format with a checksum.

$g−>cryptsetup_close ($device);

This closes an encrypted device that was created earlier by `$g->cryptsetup_open`. The `device` parameter must be the name of the mapping device (ie. */dev/mapper/mapname*) and *not* the name of the underlying block device.

This function depends on the feature `luks`. See also`$g->feature-available`.

$g−>cryptsetup_open ($device, `$key`, `$mapname` [, readonly => `$readonly`] [, crypttype => `$crypttype`]);

This command opens a block device which has been encrypted according to the Linux Unified Key Setup (LUKS) standard, Windows BitLocker, or some other types.

`device` is the encrypted block device or partition.

The caller must supply one of the keys associated with the encrypted block device, in the `key` parameter.

This creates a new block device called */dev/mapper/mapname*. Reads and writes to this block device are decrypted from and encrypted to the underlying `device` respectively.

`mapname` cannot be `"control"` because that name is reserved by device-mapper.

If the optional `crypttype` parameter is not present then libguestfs tries to guess the correct type (for example LUKS or BitLocker). However you can override this by specifying one of the following types:

`luks`
A Linux LUKS device.

`bitlk`
A Windows BitLocker device.

The optional `readonly` flag, if set to true, creates a read-only mapping.

If this block device contains LVM volume groups, then calling `$g->lvm_scan` with the `activate` parameter `true` will make them visible.

Use `$g->list_dm_devices` to list all device mapper devices.

This function depends on the feature `luks`. See also`$g->feature-available`.

$g−>dd ($src, `$dest`);

This command copies from one source device or file `src` to another destination device or file `dest`. Normally you would use this to copy to or from a device or partition, for example to duplicate a filesystem.

If the destination is a device, it must be as large or larger than the source file or device, otherwise the copy will fail. This command cannot do partial copies (see`$g->copy_device_to_device`).

This function is deprecated. In new code, use the ‘‘copy_device_to_device’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$index` = `$g`−>device_index ($device);

This function takes a device name (eg. ‘‘/dev/sdb’’) and returns the index of the device in the list of devices.

Index numbers start from 0. The named device must exist, for example as a string returned from `$g->list_devices`.

See also `$g->list_devices`, `$g->part_to_dev`.

`$output` = `$g`−>df ();

This command runs the **df** (1) command to report disk space used.

This command is mostly useful for interactive sessions. It is *not* intended that you try to parse the output string. Use `$g−>statvfs` from programs.

`$output` = `$g`−>df_h ();

This command runs the `df −h` command to report disk space used in human-readable format.

This command is mostly useful for interactive sessions. It is *not* intended that you try to parse the output string. Use `$g−>statvfs` from programs.

`$g`−>disk_create ($filename, `$format`, `$size` [, backingfile => `$backingfile`] [, backingformat => `$backingformat`] [, preallocation => `$preallocation`] [, compat => `$compat`] [, clustersize => `$clustersize`]);

Create a blank disk image called *filename* (a host file) with format `format` (usually `raw` or `qcow2`). The size is `size` bytes.

If used with the optional `backingfile` parameter, then a snapshot is created on top of the backing file. In this case,`size` must be passed as `−1`. The size of the snapshot is the same as the size of the backing file, which is discovered automatically. You are encouraged to also pass `backingformat` to describe the format of `backingfile`.

If *filename* refers to a block device, then the device is formatted. The `size` is ignored since block devices have an intrinsic size.

The other optional parameters are:

`preallocation`

If format is `raw`, then this can be either `off` (or `sparse`) or `full` to create a sparse or fully allocated file respectively. The default is `off`.

If format is `qcow2`, then this can be `off` (or `sparse`), `metadata` or `full`. Preallocating metadata can be faster when doing lots of writes, but uses more space. The default is `off`.

`compat`

`qcow2` only: Pass the string `1.1` to use the advanced qcow2 format supported by qemu X 1.1.

`clustersize`

`qcow2` only: Change the qcow2 cluster size. The default is 65536 (bytes) and this setting may be any power of two between 512 and 2097152.

Note that this call does not add the new disk to the handle. You may need to call `$g−>add_drive_opts` separately.

`$format` = `$g`−>disk_format ($filename);

Detect and return the format of the disk image called *filename*. *filename* can also be a host device, etc. If the format of the image could not be detected, then `"unknown"` is returned.

Note that detecting the disk format can be insecure under some circumstances. See "CVE−2010−3851" in **guestfs** (3).

See also: "DISK IMAGE FORMATS" in **guestfs** (3)

`$backingfile` = `$g`−>disk_has_backing_file ($filename);

Detect and return whether the disk image *filename* has a backing file.

Note that detecting disk features can be insecure under some circumstances. See "CVE−2010−3851" in **guestfs** (3).

`$size` = `$g`−>disk_virtual_size ($filename);

Detect and return the virtual size in bytes of the disk image called *filename*.

Note that detecting disk features can be insecure under some circumstances. See "CVE−2010−3851" in **guestfs** (3).

`$kmsgs` = `$g`−>dmesg ();

This returns the kernel messages (**dmesg** (1) output) from the guest kernel. This is sometimes useful for extended debugging of problems.

Another way to get the same information is to enable verbose messages with `$g−>set_verbose` or by setting the environment variable `LIBGUESTFS_DEBUG=1` before running the program.

`$g`−>download ($remotefilename, `$filename`);

Download file *remotefilename* and save it as *filename* on the local machine.

filename can also be a named pipe.

See also `$g−>upload`, `$g−>cat`.

`$g`−>download_blocks ($device, `$start`, `$stop`, `$filename` [, unallocated => `$unallocated`]);

Download the data units from *start* address to *stop* from the disk partition (eg. */dev/sda1*) and save them as *filename* on the local machine.

The use of this API on sparse disk image formats such as QCOW, may result in large zero-filled files downloaded on the host.

The size of a data unit varies across filesystem implementations. On NTFS filesystems data units are referred as clusters while on ExtX ones they are referred as fragments.

If the optional `unallocated` flag is true (default is false), only the unallocated blocks will be extracted. This is useful to detect hidden data or to retrieve deleted files which data units have not been overwritten yet.

This function depends on the feature `sleuthkit`. See also`$g−>feature−available`.

`$g`−>download_inode ($device, `$inode`, `$filename`);

Download a file given its inode from the disk partition (eg. */dev/sda1*) and save it as *filename* on the local machine.

It is not required to mount the disk to run this command.

The command is capable of downloading deleted or inaccessible files.

This function depends on the feature `sleuthkit`. See also`$g−>feature−available`.

`$g`−>download_offset ($remotefilename, `$filename`, `$offset`, `$size`);

Download file *remotefilename* and save it as *filename* on the local machine.

remotefilename is read for `size` bytes starting at `offset` (this region must be within the file or device).

Note that there is no limit on the amount of data that can be downloaded with this call, unlike with `$g−>pread`, and this call always reads the full amount unless an error occurs.

See also `$g−>download`, `$g−>pread`.

`$g`−>drop_caches ($whattodrop);

This instructs the guest kernel to drop its page cache, and/or dentries and inode caches. The parameter `whattodrop` tells the kernel what precisely to drop, see <https://linux−mm.org/Drop_Caches>

Setting `whattodrop` to 3 should drop everything.

This automatically calls **sync** (2) before the operation, so that the maximum guest memory is freed.

`$sizekb` = `$g`−>du ($path);

This command runs the `du −s` command to estimate file space usage for `path`.

`path` can be a file or a directory. If`path` is a directory then the estimate includes the contents of the directory and all subdirectories (recursively).

The result is the estimated size in *kilobytes* (ie. units of 1024 bytes).

`$g`−>e2fsck ($device [, correct => `$correct`] [, forceall => `$forceall`]);

This runs the ext2/ext3 filesystem checker on `device`. It can take the following optional arguments:

`correct`

Automatically repair the file system. This option will cause e2fsck to automatically fix any filesystem problems that can be safely fixed without human intervention.

This option may not be specified at the same time as the `forceall` option.

`forceall`

Assume an answer of XyesX to all questions; allows e2fsck to be used non-interactively.

This option may not be specified at the same time as the `correct` option.

`$g`−>e2fsck_f ($device);

This runs `e2fsck −p −f device`, ie. runs the ext2/ext3 filesystem checker on `device`, noninteractively (*−p*), even if the filesystem appears to be clean (*−f*).

This function is deprecated. In new code, use the ‘‘e2fsck’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$output` = `$g`−>echo_daemon (\@words);

This command concatenates the list of `words` passed with single spaces between them and returns the resulting string.

You can use this command to test the connection through to the daemon.

See also `$g−>ping_daemon`.

`@lines` = `$g`−>egrep ($regex, `$path`);

This calls the external **egrep** (1) program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘grep’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`−>egrepi ($regex, `$path`);

This calls the external `egrep −i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘grep’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$equality` = `$g`−>equal ($file1, `$file2`);

This compares the two files *file1* and *file2* and returns true if their content is exactly equal, or false otherwise.

The external **cmp** (1) program is used for the comparison.

`$existsflag` = `$g`−>exists ($path);

This returns `true` if and only if there is a file, directory (or anything) with the given `path` name.

See also `$g−>is_file`, `$g−>is_dir`, `$g−>stat`.

`$g`−>extlinux ($directory);

Install the SYSLINUX bootloader on the device mounted at *directory*. Unlike `$g−>syslinux` which requires a FAT filesystem, this can be used on an ext2/3/4 or btrfs filesystem.

The *directory* parameter can be either a mountpoint, or a directory within the mountpoint.

You also have to mark the partition as "active" (`$g->part_set_bootable`) and a Master Boot Record must be installed (eg. using `$g->pwrite_device`) on the first sector of the whole disk. The SYSLINUX package comes with some suitable Master Boot Records. See the **extlinux** (1) man page for further information.

Additional configuration can be supplied to SYSLINUX by placing a file called *extlinux.conf* on the filesystem under *directory*. For further information about the contents of this file, see **extlinux** (1).

See also `$g->syslinux`.

This function depends on the feature `extlinux`. See also`$g->feature-available`.

`$g`->f2fs_expand ($device);
This expands a f2fs filesystem to match the size of the underlying device.

This function depends on the feature `f2fs`. See also`$g->feature-available`.

`$g`->fallocate ($path, `$len`);
This command preallocates a file (containing zero bytes) named `path` of size `len` bytes. If the file exists already, it is overwritten.

Do not confuse this with the guestfish-specific `alloc` command which allocates a file in the host and attaches it as a device.

This function is deprecated. In new code, use the "fallocate64" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`->fallocate64 ($path, `$len`);
This command preallocates a file (containing zero bytes) named `path` of size `len` bytes. If the file exists already, it is overwritten.

Note that this call allocates disk blocks for the file. To create a sparse file use `$g->truncate_size` instead.

The deprecated call `$g->fallocate` does the same, but owing to an oversight it only allowed 30 bit lengths to be specified, effectively limiting the maximum size of files created through that call to 1GB.

Do not confuse this with the guestfish-specific `alloc` and `sparse` commands which create a file in the host and attach it as a device.

`$isavailable` = `$g`->feature_available (\@groups);
This is the same as `$g->available`, but unlike that call it returns a simple true/false boolean result, instead of throwing an exception if a feature is not found. For other documentation see `$g->available`.

`@lines` = `$g`->fgrep ($pattern, `$path`);
This calls the external **fgrep** (1) program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

This function is deprecated. In new code, use the "grep" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`->fgrepi ($pattern, `$path`);
This calls the external `fgrep -i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

This function is deprecated. In new code, use the "grep" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$description` = `$g`−>file ($path);

This call uses the standard **file** (1) command to determine the type or contents of the file.

This call will also transparently look inside various types of compressed file.

The exact command which runs is `file −zb path`. Note in particular that the filename is not prepended to the output (the *−b* option).

The output depends on the output of the underlying **file** (1) command and it can change in future in ways beyond our control. In other words, the output is not guaranteed by the ABI.

See also: **file** (1), `$g−>vfs_type`, `$g−>lstat`, `$g−>is_file`, `$g−>is_blockdev` (etc), `$g−>is_zero`.

`$arch` = `$g`−>file_architecture ($filename);

This detects the architecture of the binary *filename*, and returns it if known.

Currently defined architectures are:

"aarch64"
: 64 bit ARM.

"arm"
: 32 bit ARM.

"i386"
: This string is returned for all 32 bit i386, i486, i586, i686 binaries irrespective of the precise processor requirements of the binary.

"ia64"
: Intel Itanium.

"ppc"
: 32 bit Power PC.

"ppc64"
: 64 bit Power PC (big endian).

"ppc64le"
: 64 bit Power PC (little endian).

"riscv32"
"riscv64"
"riscv128"
: RISC-V 32−, 64− or 128−bit variants.

"s390"
: 31 bit IBM S/390.

"s390x"
: 64 bit IBM S/390.

"sparc"
: 32 bit SPARC.

"sparc64"
: 64 bit SPARC V9 and above.

"x86_64"
: 64 bit x86−64.

Libguestfs may return other architecture strings in future.

The function works on at least the following types of files:

- many types of Un*x and Linux binary
- many types of Un*x and Linux shared library
- Windows Win32 and Win64 binaries
- Windows Win32 and Win64 DLLs

  Win32 binaries and DLLs return `i386`.

  Win64 binaries and DLLs return `x86_64`.
- Linux kernel modules
- Linux new-style initrd images
- some non−x86 Linux vmlinuz kernels

What it can't do currently:

- static libraries (libfoo.a)
- Linux old-style initrd as compressed ext2 filesystem (RHEL 3)
- x86 Linux vmlinuz kernels

  x86 vmlinuz images (bzImage format) consist of a mix of 16−, 32− and compressed code, and are horribly hard to unpack. If you want to find the architecture of a kernel, use the architecture of the associated initrd or kernel module(s) instead.

`$size` = `$g`−>filesize ($file);

This command returns the size of *file* in bytes.

To get other stats about a file, use `$g−>stat`, `$g−>lstat`, `$g−>is_dir`, `$g−>is_file` etc. To get the size of block devices, use `$g−>blockdev_getsize64`.

`$fsavail` = `$g`−>filesystem_available ($filesystem);

Check whether libguestfs supports the named filesystem. The argument `filesystem` is a filesystem name, such as `ext3`.

You must call `$g−>launch` before using this command.

This is mainly useful as a negative test. If this returns true, it doesn't mean that a particular filesystem can be created or mounted, since filesystems can fail for other reasons such as it being a later version of the filesystem, or having incompatible features, or lacking the right mkfs.<*fs*> tool.

See also `$g−>available`, `$g−>feature_available`, "AVAILABILITY" in **guestfs**(3).

`@dirents` = `$g`−>filesystem_walk ($device);

Walk through the internal structures of a disk partition (eg. */dev/sda1*) in order to return a list of all the files and directories stored within.

It is not necessary to mount the disk partition to run this command.

All entries in the filesystem are returned. This function can list deleted or unaccessible files. The entries are *not* sorted.

The `tsk_dirent` structure contains the following fields.

`tsk_inode`

Filesystem reference number of the node. It might be `0` if the node has been deleted.

`tsk_type`

Basic file type information. See below for a detailed list of values.

`tsk_size`

File size in bytes. It might be `−1` if the node has been deleted.

`tsk_name`
: The file path relative to its directory.

`tsk_flags`
: Bitfield containing extra information regarding the entry. It contains the logical OR of the following values:

  0x0001
  : If set to `1`, the file is allocated and visible within the filesystem. Otherwise, the file has been deleted. Under certain circumstances, the function`download_inode` can be used to recover deleted files.

  0x0002
  : Filesystem such as NTFS and Ext2 or greater, separate the file name from the metadata structure. The bit is set to`1` when the file name is in an unallocated state and the metadata structure is in an allocated one. This generally implies the metadata has been reallocated to a new file. Therefore, information such as file type, file size, timestamps, number of links and symlink target might not correspond with the ones of the original deleted entry.

  0x0004
  : The bit is set to `1` when the file is compressed using filesystem native compression support (NTFS). The API is not able to detect application level compression.

`tsk_atime_sec`
`tsk_atime_nsec`
`tsk_mtime_sec`
`tsk_mtime_nsec`
`tsk_ctime_sec`
`tsk_ctime_nsec`
`tsk_crtime_sec`
`tsk_crtime_nsec`
: Respectively, access, modification, last status change and creation time in Unix format in seconds and nanoseconds.

`tsk_nlink`
: Number of file names pointing to this entry.

`tsk_link`
: If the entry is a symbolic link, this field will contain the path to the target file.

The `tsk_type` field will contain one of the following characters:

'b'	Block special
'c'	Char special
'd'	Directory
'f'	FIFO (named pipe)
'l'	Symbolic link
'r'	Regular file
's'	Socket
'h'	Shadow inode (Solaris)
'w'	Whiteout inode (BSD)
'u'	Unknown file type

This function depends on the feature `libtsk`. See also`$g->feature-available`.

$g->fill ($c, $len, $path);
> This command creates a new file called `path`. The initial content of the file is `len` octets of `c`, where `c` must be a number in the range `[0..255]`.
>
> To fill a file with zero bytes (sparsely), it is much more efficient to use `$g->truncate_size`. To create a file with a pattern of repeating bytes use `$g->fill_pattern`.

$g->fill_dir ($dir, $nr);
> This function, useful for testing filesystems, creates `nr` empty files in the directory `dir` with names `00000000` through `nr-1` (ie. each file name is 8 digits long padded with zeroes).

$g->fill_pattern ($pattern, $len, $path);
> This function is like `$g->fill` except that it creates a new file of length `len` containing the repeating pattern of bytes in `pattern`. The pattern is truncated if necessary to ensure the length of the file is exactly `len` bytes.

@names = $g->find ($directory);
> This command lists out all files and directories, recursively, starting at *directory*. It is essentially equivalent to running the shell command `find directory -print` but some post-processing happens on the output, described below.
>
> This returns a list of strings *without any prefix*. Thus if the directory structure was:

```
 /tmp/a
 /tmp/b
 /tmp/c/d
```

> then the returned list from `$g->find` */tmp* would be 4 elements:

```
 a
 b
 c
 c/d
```

> If *directory* is not a directory, then this command returns an error.
>
> The returned list is sorted.

$g->find0 ($directory, $files);
> This command lists out all files and directories, recursively, starting at *directory*, placing the resulting list in the external file called *files*.
>
> This command works the same way as `$g->find` with the following exceptions:
>
> - The resulting list is written to an external file.
> - Items (filenames) in the result are separated by `\0` characters. See **find** (1) option *−print0*.
> - The result list is not sorted.

@dirents = $g->find_inode ($device, $inode);
> Searches all the entries associated with the given inode.
>
> For each entry, a `tsk_dirent` structure is returned. See `filesystem_walk` for more information about `tsk_dirent` structures.
>
> This function depends on the feature `libtsk`. See also`$g->feature-available`.

$device = $g->findfs_label ($label);
> This command searches the filesystems and returns the one which has the given label. An error is returned if no such filesystem can be found.
>
> To find the label of a filesystem, use `$g->vfs_label`.

`$device` = `$g`−>findfs_uuid ($uuid);

This command searches the filesystems and returns the one which has the given UUID. An error is returned if no such filesystem can be found.

To find the UUID of a filesystem, use `$g->vfs_uuid`.

`$status` = `$g`−>fsck ($fstype, `$device`);

This runs the filesystem checker (fsck) on `device` which should have filesystem type `fstype`.

The returned integer is the status. See **fsck** (8) for the list of status codes from `fsck`.

Notes:

- Multiple status codes can be summed together.
- A non-zero return code can mean "success", for example if errors have been corrected on the filesystem.
- Checking or repairing NTFS volumes is not supported (by linux-ntfs).

This command is entirely equivalent to running `fsck -a -t fstype device`.

`$g`−>fstrim ($mountpoint [, offset => `$offset`] [, length => `$length`] [, minimumfreeextent => `$minimumfreeextent`]);

Trim the free space in the filesystem mounted on `mountpoint`. The filesystem must be mounted read-write.

The filesystem contents are not affected, but any free space in the filesystem is "trimmed", that is, given back to the host device, thus making disk images more sparse, allowing unused space in qcow2 files to be reused, etc.

This operation requires support in libguestfs, the mounted filesystem, the host filesystem, qemu and the host kernel. If this support isn't present it may give an error or even appear to run but do nothing.

In the case where the kernel vfs driver does not support trimming, this call will fail with errno set to `ENOTSUP`. Currently this happens when trying to trim FAT filesystems.

See also `$g->zero_free_space`. That is a slightly different operation that turns free space in the filesystem into zeroes. It is valid to call `$g->fstrim` either instead of, or after calling `$g->zero_free_space`.

This function depends on the feature `fstrim`. See also`$g->feature-available`.

`$append` = `$g`−>get_append ();

Return the additional kernel options which are added to the libguestfs appliance kernel command line.

If `NULL` then no options are added.

`$backend` = `$g`−>get_attach_method ();

Return the current backend.

See `$g->set_backend` and "BACKEND" in **guestfs** (3).

This function is deprecated. In new code, use the "get_backend" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$autosync` = `$g`−>get_autosync ();

Get the autosync flag.

`$backend` = `$g`−>get_backend ();

Return the current backend.

This handle property was previously called the "attach method".

See `$g->set_backend` and "BACKEND" in **guestfs** (3).

`$val` = `$g`−>get_backend_setting ($name);

Find a backend setting string which is either `"name"` or begins with `"name="`. If`"name"`, this returns the string `"1"`. If`"name="`, this returns the part after the equals sign (which may be an empty string).

If no such setting is found, this function throws an error. The errno (see`$g->last_errno`) will be `ESRCH` in this case.

See "BACKEND" in **guestfs** (3), "BACKEND SETTINGS" in **guestfs** (3).

`@settings` = `$g`−>get_backend_settings ();

Return the current backend settings.

This call returns all backend settings strings. If you want to find a single backend setting, see `$g->get_backend_setting`.

See "BACKEND" in **guestfs** (3), "BACKEND SETTINGS" in **guestfs** (3).

`$cachedir` = `$g`−>get_cachedir ();

Get the directory used by the handle to store the appliance cache.

`$direct` = `$g`−>get_direct ();

Return the direct appliance mode flag.

This function is deprecated. In new code, use the "internal_get_console_socket" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$attrs` = `$g`−>get_e2attrs ($file);

This returns the file attributes associated with *file*.

The attributes are a set of bits associated with each inode which affect the behaviour of the file. The attributes are returned as a string of letters (described below). The string may be empty, indicating that no file attributes are set for this file.

These attributes are only present when the file is located on an ext2/3/4 filesystem. Using this call on other filesystem types will result in an error.

The characters (file attributes) in the returned string are currently:

’A’	When the file is accessed, its atime is not modified.
’a’	The file is append-only.
’c’	The file is compressed on-disk.
’D’	(Directories only.) Changes to this directory are written synchronously to disk.
’d’	The file is not a candidate for backup (see **dump** (8)).
’E’	The file has compression errors.
’e’	The file is using extents.
’h’	The file is storing its blocks in units of the filesystem blocksize instead of sectors.
’I’	(Directories only.) The directory is using hashed trees.
’i’	The file is immutable. It cannot be modified, deleted or renamed. No link can be created to this file.
’j’	The file is data-journaled.
’s’	When the file is deleted, all its blocks will be zeroed.
’S’	Changes to this file are written synchronously to disk.
’T’	(Directories only.) This is a hint to the block allocator that subdirectories contained in this directory should be spread across blocks. If not present, the block allocator will try to group

subdirectories together.

’t’ For a file, this disables tail-merging. (Not used by upstream implementations of ext2.)

’u’ When the file is deleted, its blocks will be saved, allowing the file to be undeleted.

’X’ The raw contents of the compressed file may be accessed.

’Z’ The compressed file is dirty.

More file attributes may be added to this list later. Not all file attributes may be set for all kinds of files. For detailed information, consult the **chattr** (1) man page.

See also `$g−>set_e2attrs`.

Don’t confuse these attributes with extended attributes (see `$g−>getxattr`).

`$generation` = `$g`−>get_e2generation ($file);
This returns the ext2 file generation of a file. The generation (which used to be called the ‘‘version’’) is a number associated with an inode. This is most commonly used by NFS servers.

The generation is only present when the file is located on an ext2/3/4 filesystem. Using this call on other filesystem types will result in an error.

See `$g−>set_e2generation`.

`$label` = `$g`−>get_e2label ($device);
This returns the ext2/3/4 filesystem label of the filesystem on `device`.

This function is deprecated. In new code, use the ‘‘vfs_label’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$uuid` = `$g`−>get_e2uuid ($device);
This returns the ext2/3/4 filesystem UUID of the filesystem on `device`.

This function is deprecated. In new code, use the ‘‘vfs_uuid’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$hv` = `$g`−>get_hv ();
Return the current hypervisor binary.

This is always non-NULL. If it wasn’t set already, then this will return the default qemu binary name.

`$identifier` = `$g`−>get_identifier ();
Get the handle identifier. See`$g−>set_identifier`.

`$challenge` = `$g`−>get_libvirt_requested_credential_challenge ($index);
Get the challenge (provided by libvirt) for the `index`’th requested credential. If libvirt did not provide a challenge, this returns the empty string `""`.

See ‘‘LIBVIRT AUTHENTICATION’’ in **guestfs** (3) for documentation and example code.

`$defresult` = `$g`−>get_libvirt_requested_credential_defresult ($index);
Get the default result (provided by libvirt) for the `index`’th requested credential. If libvirt did not provide a default result, this returns the empty string `""`.

See ‘‘LIBVIRT AUTHENTICATION’’ in **guestfs** (3) for documentation and example code.

`$prompt` = `$g`−>get_libvirt_requested_credential_prompt ($index);
Get the prompt (provided by libvirt) for the `index`’th requested credential. If libvirt did not provide a prompt, this returns the empty string `""`.

See ‘‘LIBVIRT AUTHENTICATION’’ in **guestfs** (3) for documentation and example code.

`@creds` = `$g`−>get_libvirt_requested_credentials ();
: This should only be called during the event callback for events of type `GUESTFS_EVENT_LIBVIRT_AUTH`.

  Return the list of credentials requested by libvirt. Possible values are a subset of the strings provided when you called `$g->set_libvirt_supported_credentials`.

  See ''LIBVIRT AUTHENTICATION'' in **guestfs** (3) for documentation and example code.

`$memsize` = `$g`−>get_memsize ();
: This gets the memory size in megabytes allocated to the hypervisor.

  If `$g->set_memsize` was not called on this handle, and if `LIBGUESTFS_MEMSIZE` was not set, then this returns the compiled-in default value for memsize.

  For more information on the architecture of libguestfs, see **guestfs** (3).

`$network` = `$g`−>get_network ();
: This returns the enable network flag.

`$path` = `$g`−>get_path ();
: Return the current search path.

  This is always non-NULL. If it wasn't set already, then this will return the default path.

`$pgroup` = `$g`−>get_pgroup ();
: This returns the process group flag.

`$pid` = `$g`−>get_pid ();
: Return the process ID of the hypervisor. If there is no hypervisor running, then this will return an error.

  This is an internal call used for debugging and testing.

`$program` = `$g`−>get_program ();
: Get the program name. See `$g->set_program`.

`$hv` = `$g`−>get_qemu ();
: Return the current hypervisor binary (usually qemu).

  This is always non-NULL. If it wasn't set already, then this will return the default qemu binary name.

  This function is deprecated. In new code, use the ''get_hv'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$recoveryproc` = `$g`−>get_recovery_proc ();
: Return the recovery process enabled flag.

`$selinux` = `$g`−>get_selinux ();
: This returns the current setting of the selinux flag which is passed to the appliance at boot time. See `$g->set_selinux`.

  For more information on the architecture of libguestfs, see **guestfs** (3).

  This function is deprecated. In new code, use the ''selinux_relabel'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$smp` = `$g`−>get_smp ();
: This returns the number of virtual CPUs assigned to the appliance.

`$sockdir` = `$g`−>get_sockdir ();
: Get the directory used by the handle to store temporary socket files.

  This is different from `$g->get_tmpdir`, as we need shorter paths for sockets (due to the limited buffers of filenames for UNIX sockets), and `$g->get_tmpdir` may be too long for them.

The environment variable `XDG_RUNTIME_DIR` controls the default value: If `XDG_RUNTIME_DIR` is set, then that is the default. Else */tmp* is the def ault.

`$state` = `$g`−>get_state ();

This returns the current state as an opaque integer. This is only useful for printing debug and internal error messages.

For more information on states, see **guestfs** (3).

`$tmpdir` = `$g`−>get_tmpdir ();

Get the directory used by the handle to store temporary files.

`$trace` = `$g`−>get_trace ();

Return the command trace flag.

`$mask` = `$g`−>get_umask ();

Return the current umask. By default the umask is `022` unless it has been set by calling `$g->umask`.

`$verbose` = `$g`−>get_verbose ();

This returns the verbose messages flag.

`$context` = `$g`−>getcon ();

This gets the SELinux security context of the daemon.

See the documentation about SELINUX in **guestfs** (3), and `$g->setcon`

This function depends on the feature `selinux`. See also`$g->feature-available`.

This function is deprecated. In new code, use the ''selinux_relabel'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$xattr` = `$g`−>getxattr ($path, `$name`);

Get a single extended attribute from file `path` named `name`. This call follows symlinks. If you want to lookup an extended attribute for the symlink itself, use `$g->lgetxattr`.

Normally it is better to get all extended attributes from a file in one go by calling `$g->getxattrs`. However some Linux filesystem implementations are buggy and do not provide a way to list out attributes. For these filesystems (notably ntfs−3g) you have to know the names of the extended attributes you want in advance and call this function.

Extended attribute values are blobs of binary data. If there is no extended attribute named `name`, this returns an error.

See also: `$g->getxattrs`, `$g->lgetxattr`, **attr** (5).

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`@xattrs` = `$g`−>getxattrs ($path);

This call lists the extended attributes of the file or directory `path`.

At the system call level, this is a combination of the **listxattr** (2) and **getxattr** (2) calls.

See also: `$g->lgetxattrs`, **attr** (5).

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`@paths` = `$g`−>glob_expand ($pattern [, directoryslash => `$directoryslash`]);

This command searches for all the pathnames matching `pattern` according to the wildcard expansion rules used by the shell.

If no paths match, then this returns an empty list (note: not an error).

It is just a wrapper around the C **glob** (3) function with flags `GLOB_MARK|GLOB_BRACE`. See that manual page for more details.

`directoryslash` controls whether use the `GLOB_MARK` flag for **glob** (3), and it defaults to true. It can be explicitly set as off to return no trailing slashes in filenames of directories.

Notice that there is no equivalent command for expanding a device name (eg. */dev/sd**). Use `$g->list_devices`, `$g->list_partitions` etc functions instead.

`@paths` = `$g`−>glob_expand_opts ($pattern [, directoryslash => `$directoryslash`]);

This is an alias of ‘‘glob_expand’’.

`@lines` = `$g`−>grep ($regex, `$path` [, extended => `$extended`] [, fixed => `$fixed`] [, insensitive => `$insensitive`] [, compressed => `$compressed`]);

This calls the external **grep** (1) program and returns the matching lines.

The optional flags are:

`extended`

Use extended regular expressions. This is the same as using the *−E* flag.

`fixed`

Match fixed (don’t use regular expressions). This is the same as using the *−F* flag.

`insensitive`

Match case-insensitive. This is the same as using the *−i* flag.

`compressed`

Use **zgrep** (1) instead of **grep** (1). This allows the input to be compress− or gzip-compressed.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

`@lines` = `$g`−>grep_opts ($regex, `$path` [, extended => `$extended`] [, fixed => `$fixed`] [, insensitive => `$insensitive`] [, compressed => `$compressed`]);

This is an alias of ‘‘grep’’.

`@lines` = `$g`−>grepi ($regex, `$path`);

This calls the external `grep −i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘grep’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>grub_install ($root, `$device`);

This command installs GRUB 1 (the Grand Unified Bootloader) on `device`, with the root directory being `root`.

Notes:

- There is currently no way in the API to install grub2, which is used by most modern Linux guests. It is possible to run the grub2 command from the guest, although see the caveats in ‘‘RUNNING COMMANDS’’ in **guestfs** (3).
- This uses **grub−install** (8) from the host. Unfortunately grub is not always compatible with itself, so this only works in rather narrow circumstances. Careful testing with each guest version is advisable.
- If grub-install reports the error ‘‘No suitable drive was found in the generated device map.’’ it may be that you need to create a */boot/grub/device.map* file first that contains the mapping between grub device names and Linux device names. It is usually sufficient to create a file containing:

```
 (hd0) /dev/vda
```

replacing */dev/vda* with the name of the installation device.

This function depends on the feature `grub`. See also`$g->feature-available`.

@lines = $g->head ($path);
: This command returns up to the first 10 lines of a file as a list of strings.

  Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

@lines = $g->head_n ($nrlines, $path);
: If the parameter `nrlines` is a positive number, this returns the first `nrlines` lines of the file `path`.

  If the parameter `nrlines` is a negative number, this returns lines from the file `path`, excluding the last `nrlines` lines.

  If the parameter `nrlines` is zero, this returns an empty list.

  Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

$dump = $g->hexdump ($path);
: This runs `hexdump -C` on the given `path`. The result is the human-readable, canonical hex dump of the file.

  Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

$g->hivex_close ();
: Close the current hivex handle.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

$g->hivex_commit ($filename);
: Commit (write) changes to the hive.

  If the optional *filename* parameter is null, then the changes are written back to the same hive that was opened. If this is not null then they are written to the alternate filename given and the original hive is left untouched.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

$nodeh = $g->hivex_node_add_child ($parent, $name);
: Add a child node to `parent` named `name`.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

@nodehs = $g->hivex_node_children ($nodeh);
: Return the list of nodes which are subkeys of `nodeh`.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

$g->hivex_node_delete_child ($nodeh);
: Delete `nodeh`, recursively if necessary.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

`$child` = `$g`−>hivex_node_get_child ($nodeh, `$name`);

Return the child of `nodeh` with the name `name`, if it exists. This can return `0` meaning the name was not found.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$valueh` = `$g`−>hivex_node_get_value ($nodeh, `$key`);

Return the value attached to `nodeh` which has the name `key`, if it exists. This can return `0` meaning the key was not found.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$name` = `$g`−>hivex_node_name ($nodeh);

Return the name of `nodeh`.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$parent` = `$g`−>hivex_node_parent ($nodeh);

Return the parent node of `nodeh`.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$g`−>hivex_node_set_value ($nodeh, `$key`, `$t`, `$val`);

Set or replace a single value under the node `nodeh`. The `key` is the name, `t` is the type, and `val` is the data.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`@valuehs` = `$g`−>hivex_node_values ($nodeh);

Return the array of (key, datatype, data) tuples attached to `nodeh`.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$g`−>hivex_open ($filename [, verbose => `$verbose`] [, debug => `$debug`] [, write => `$write`] [, unsafe => `$unsafe`]);

Open the Windows Registry hive file named *filename*. If there was any previous hivex handle associated with this guestfs session, then it is closed.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$nodeh` = `$g`−>hivex_root ();

Return the root node of the hive.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$key` = `$g`−>hivex_value_key ($valueh);

Return the key (name) field of a (key, datatype, data) tuple.

This is a wrapper around the **hivex** (3) call of the same name.

This function depends on the feature `hivex`. See also `$g−>feature−available`.

`$databuf` = `$g`−>hivex_value_string ($valueh);
: This calls `$g->hivex_value_value` (which returns the data field from a hivex value tuple). It then assumes that the field is a UTF−16LE string and converts the result to UTF−8 (or if this is not possible, it returns an error).

  This is useful for reading strings out of the Windows registry. However it is not foolproof because the registry is not strongly-typed and fields can contain arbitrary or unexpected data.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

`$datatype` = `$g`−>hivex_value_type ($valueh);
: Return the data type field from a (key, datatype, data) tuple.

  This is a wrapper around the **hivex** (3) call of the same name.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

`$databuf` = `$g`−>hivex_value_utf8 ($valueh);
: This calls `$g->hivex_value_value` (which returns the data field from a hivex value tuple). It then assumes that the field is a UTF−16LE string and converts the result to UTF−8 (or if this is not possible, it returns an error).

  This is useful for reading strings out of the Windows registry. However it is not foolproof because the registry is not strongly-typed and fields can contain arbitrary or unexpected data.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

  This function is deprecated. In new code, use the ''hivex_value_string'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$databuf` = `$g`−>hivex_value_value ($valueh);
: Return the data field of a (key, datatype, data) tuple.

  This is a wrapper around the **hivex** (3) call of the same name.

  See also: `$g->hivex_value_utf8`.

  This function depends on the feature `hivex`. See also`$g->feature-available`.

`$content` = `$g`−>initrd_cat ($initrdpath, `$filename`);
: This command unpacks the file *filename* from the initrd file called *initrdpath*. The filename must be given *without* the initial */* character.

  For example, in guestfish you could use the following command to examine the boot script (usually called */init*) contained in a Linux initrd or initramfs image:

```
 initrd-cat /boot/initrd-<version>.img init
```

  See also `$g->initrd_list`.

  Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

`@filenames` = `$g`−>initrd_list ($path);
: This command lists out files contained in an initrd.

  The files are listed without any initial */* character. The files are listed in the order they appear (not necessarily alphabetical). Directory names are listed as separate items.

  Old Linux kernels (2.4 and earlier) used a compressed ext2 filesystem as initrd. We *only* support the newer initramfs format (compressed cpio files).

`$wd` = `$g`−>inotify_add_watch ($path, `$mask`);
: Watch `path` for the events listed in `mask`.

Note that if `path` is a directory then events within that directory are watched, but this does *not* happen recursively (in subdirectories).

Note for non-C or non-Linux callers: the inotify events are defined by the Linux kernel ABI and are listed in */usr/include/sys/inotify.h*.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`$g`−>inotify_close ();

This closes the inotify handle which was previously opened by inotify_init. It removes all watches, throws away any pending events, and deallocates all resources.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`@paths` = `$g`−>inotify_files ();

This function is a helpful wrapper around `$g->inotify_read` which just returns a list of pathnames of objects that were touched. The returned pathnames are sorted and deduplicated.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`$g`−>inotify_init ($maxevents);

This command creates a new inotify handle. The inotify subsystem can be used to notify events which happen to objects in the guest filesystem.

`maxevents` is the maximum number of events which will be queued up between calls to `$g->inotify_read` or `$g->inotify_files`. If this is passed as`0`, then the k ernel (or previously set) default is used. For Linux 2.6.29 the default was 16384 events. Beyond this limit, the kernel throws away events, but records the fact that it threw them away by setting a flag `IN_Q_OVERFLOW` in the returned structure list (see `$g->inotify_read`).

Before any events are generated, you have to add some watches to the internal watch list. See: `$g->inotify_add_watch` and `$g->inotify_rm_watch`.

Queued up events should be read periodically by calling `$g->inotify_read` (or `$g->inotify_files` which is just a helpful wrapper around `$g->inotify_read`). If you don't read the events out often enough then you risk the internal queue overflowing.

The handle should be closed after use by calling `$g->inotify_close`. This also removes any watches automatically.

See also **inotify** (7) for an overview of the inotify interface as exposed by the Linux kernel, which is roughly what we expose via libguestfs. Note that there is one global inotify handle per libguestfs instance.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`@events` = `$g`−>inotify_read ();

Return the complete queue of events that have happened since the previous read call.

If no events have happened, this returns an empty list.

Note: In order to make sure that all events have been read, you must call this function repeatedly until it returns an empty list. The reason is that the call will read events up to the maximum appliance-to-host message size and leave remaining events in the queue.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`$g`−>inotify_rm_watch ($wd);

Remove a previously defined inotify watch. See`$g->inotify_add_watch`.

This function depends on the feature `inotify`. See also`$g->feature-available`.

`$arch` = `$g`−>inspect_get_arch ($root);

This returns the architecture of the inspected operating system. The possible return values are listed under `$g->file_architecture`.

If the architecture could not be determined, then the string `unknown` is returned.

Please read "INSPECTION" in **guestfs** (3) for more details.

`$distro` = `$g`−>inspect_get_distro ($root);

This returns the distro (distribution) of the inspected operating system.

Currently defined distros are:

"alpinelinux"
: Alpine Linux.

"altlinux"
: ALT Linux.

"archlinux"
: Arch Linux.

"buildroot"
: Buildroot-derived distro, but not one we specifically recognize.

"centos"
: CentOS.

"cirros"
: Cirros.

"coreos"
: CoreOS.

"debian"
: Debian.

"fedora"
: Fedora.

"freebsd"
: FreeBSD.

"freedos"
: FreeDOS.

"frugalware"
: Frugalware.

"gentoo"
: Gentoo.

"kalilinux"
: Kali Linux.

"linuxmint"
: Linux Mint.

"mageia"
: Mageia.

"mandriva"
: Mandriva.

"meego"
: MeeGo.

"msdos"
: Microsoft DOS.

"neokylin"
NeoKylin.

"netbsd"
NetBSD.

"openbsd"
OpenBSD.

"openmandriva"
OpenMandriva Lx.

"opensuse"
OpenSUSE.

"oraclelinux"
Oracle Linux.

"pardus"
Pardus.

"pldlinux"
PLD Linux.

"redhat-based"
Some Red Hat-derived distro.

"rhel"
Red Hat Enterprise Linux.

"scientificlinux"
Scientific Linux.

"slackware"
Slackware.

"sles"
SuSE Linux Enterprise Server or Desktop.

"suse-based"
Some openSuSE-derived distro.

"ttylinux"
ttylinux.

"ubuntu"
Ubuntu.

"unknown"
The distro could not be determined.

"voidlinux"
Void Linux.

"windows"
Windows does not have distributions. This string is returned if the OS type is Windows.

Future versions of libguestfs may return other strings here. The caller should be prepared to handle any string.

Please read "INSPECTION" in **guestfs** (3) for more details.

`%drives` = `$g`->inspect_get_drive_mappings ($root);
This call is useful for Windows which uses a primitive system of assigning drive letters (like *C:\*) to partitions. This inspection API examines the Windows Registry to find out how disks/partitions are mapped to drive letters, and returns a hash table as in the example below:

```
 C      =>     /dev/vda2
 E      =>     /dev/vdb1
 F      =>     /dev/vdc1
```

Note that keys are drive letters. For Windows, the key is case insensitive and just contains the drive letter, without the customary colon separator character.

In future we may support other operating systems that also used drive letters, but the keys for those might not be case insensitive and might be longer than 1 character. For example in OS−9, hard drives were named `h0`, `h1` etc.

For Windows guests, currently only hard drive mappings are returned. Removable disks (eg. DVD-ROMs) are ignored.

For guests that do not use drive mappings, or if the drive mappings could not be determined, this returns an empty hash table.

Please read ''INSPECTION'' in **guestfs**(3) for more details. See also `$g−>inspect_get_mountpoints`, `$g−>inspect_get_filesystems`.

`@filesystems` = `$g`−>inspect_get_filesystems ($root);

This returns a list of all the filesystems that we think are associated with this operating system. This includes the root filesystem, other ordinary filesystems, and non-mounted devices like swap partitions.

In the case of a multi-boot virtual machine, it is possible for a filesystem to be shared between operating systems.

Please read ''INSPECTION'' in **guestfs**(3) for more details. See also `$g−>inspect_get_mountpoints`.

`$format` = `$g`−>inspect_get_format ($root);

Before libguestfs 1.38, there was some unreliable support for detecting installer CDs. This API would return:

`installed`
This is an installed operating system.

`installer`
The disk image being inspected is not an installed operating system, but a *bootable* install disk, live CD, or similar.

`unknown`
The format of this disk image is not known.

In libguestfs X 1.38, this only returns `installed`. Use libosinfo directly to detect installer CDs.

Please read ''INSPECTION'' in **guestfs**(3) for more details.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs**(3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$hostname` = `$g`−>inspect_get_hostname ($root);

This function returns the hostname of the operating system as found by inspection of the guestXs configuration files.

If the hostname could not be determined, then the string `unknown` is returned.

Please read ''INSPECTION'' in **guestfs**(3) for more details.

`$icon` = `$g`−>inspect_get_icon ($root [, favicon => `$favicon`] [, highquality => `$highquality`]);

This function returns an icon corresponding to the inspected operating system. The icon is returned as a buffer containing a PNG image (re-encoded to PNG if necessary).

If it was not possible to get an icon this function returns a zero-length (non-NULL) buffer. *Callers must check for this case*.

Libguestfs will start by looking for a file called */etc/favicon.png* or *C:\etc\favicon.png* and if it has the correct format, the contents of this file will be returned. You can disable favicons by passing the optional `favicon` boolean as false (default is true).

If finding the favicon fails, then we look in other places in the guest for a suitable icon.

If the optional `highquality` boolean is true then only high quality icons are returned, which means only icons of high resolution with an alpha channel. The default (false) is to return any icon we can, even if it is of substandard quality.

Notes:

- Unlike most other inspection API calls, the guestXs disks must be mounted up before you call this, since it needs to read information from the guest filesystem during the call.
- **Security:** The icon data comes from the untrusted guest, and should be treated with caution. PNG files have been known to contain exploits. Ensure that libpng (or other relevant libraries) are fully up to date before trying to process or display the icon.
- The PNG image returned can be any size. It might not be square. Libguestfs tries to return the largest, highest quality icon available. The application must scale the icon to the required size.
- Extracting icons from Windows guests requires the external **wrestool**(1) program from the `icoutils` package, and several programs (**bmptopnm**(1), **pnmtopng**(1), **pamcut**(1)) from the `netpbm` package. These must be installed separately.
- Operating system icons are usually trademarks. Seek legal advice before using trademarks in applications.

`$major` = `$g`−>inspect_get_major_version ($root);

This returns the major version number of the inspected operating system.

Windows uses a consistent versioning scheme which is *not* reflected in the popular public names used by the operating system. Notably the operating system known as ‘‘Windows 7’’ is really version 6.1 (ie. major = 6, minor = 1). You can find out the real versions corresponding to releases of Windows by consulting Wikipedia or MSDN.

If the version could not be determined, then `0` is returned.

Please read ‘‘INSPECTION’’ in **guestfs**(3) for more details.

`$minor` = `$g`−>inspect_get_minor_version ($root);

This returns the minor version number of the inspected operating system.

If the version could not be determined, then `0` is returned.

Please read ‘‘INSPECTION’’ in **guestfs**(3) for more details. See also `$g−>inspect_get_major_version`.

`%mountpoints` = `$g`−>inspect_get_mountpoints ($root);

This returns a hash of where we think the filesystems associated with this operating system should be mounted. Callers should note that this is at best an educated guess made by reading configuration files such as */etc/fstab*. *In particular note* that this may return filesystems which are non-existent or not mountable and callers should be prepared to handle or ignore failures if they try to mount them.

Each element in the returned hashtable has a key which is the path of the mountpoint (eg. */boot*) and a value which is the filesystem that would be mounted there (eg. */dev/sda1*).

Non-mounted devices such as swap devices are *not* returned in this list.

For operating systems like Windows which still use drive letters, this call will only return an entry for the first drive ‘‘mounted on’’ */*. For information about the mapping of drive letters to partitions, see

`$g->inspect_get_drive_mappings`.

Please read "INSPECTION" in **guestfs**(3) for more details. See also `$g->inspect_get_filesystems`.

`$id` = `$g`->inspect_get_osinfo ($root);

This function returns a possible short ID for libosinfo corresponding to the guest.

Note: The returned ID is only a guess by libguestfs, and nothing ensures that it actually exists in osinfo-db.

If no ID could not be determined, then the string `unknown` is returned.

`$packageformat` = `$g`->inspect_get_package_format ($root);

This function and `$g->inspect_get_package_management` return the package format and package management tool used by the inspected operating system. For example for Fedora these functions would return `rpm` (package format), and `yum` or `dnf` (package management).

This returns the string `unknown` if we could not determine the package format *or* if the operating system does not have a real packaging system (eg. Windows).

Possible strings include: `rpm`, `deb`, `ebuild`, `pisi`, `pacman`, `pkgsrc`, `apk`, `xbps`. Future versions of libguestfs may return other strings.

Please read "INSPECTION" in **guestfs**(3) for more details.

`$packagemanagement` = `$g`->inspect_get_package_management ($root);

`$g->inspect_get_package_format` and this function return the package format and package management tool used by the inspected operating system. For example for Fedora these functions would return `rpm` (package format), and `yum` or `dnf` (package management).

This returns the string `unknown` if we could not determine the package management tool *or* if the operating system does not have a real packaging system (eg. Windows).

Possible strings include: `yum`, `dnf`, `up2date`, `apt` (for all Debian derivatives), `portage`, `pisi`, `pacman`, `urpmi`, `zypper`, `apk`, `xbps`. Future versions of libguestfs may return other strings.

Please read "INSPECTION" in **guestfs**(3) for more details.

`$product` = `$g`->inspect_get_product_name ($root);

This returns the product name of the inspected operating system. The product name is generally some freeform string which can be displayed to the user, but should not be parsed by programs.

If the product name could not be determined, then the string `unknown` is returned.

Please read "INSPECTION" in **guestfs**(3) for more details.

`$variant` = `$g`->inspect_get_product_variant ($root);

This returns the product variant of the inspected operating system.

For Windows guests, this returns the contents of the Registry key `HKLM\Software\Microsoft\Windows NT\CurrentVersion InstallationType` which is usually a string such as `Client` or `Server` (other values are possible). This can be used to distinguish consumer and enterprise versions of Windows that have the same version number (for example, Windows 7 and Windows 2008 Server are both version 6.1, but the former is `Client` and the latter is `Server`).

For enterprise Linux guests, in future we intend this to return the product variant such as `Desktop`, `Server` and so on. But this is not implemented at present.

If the product variant could not be determined, then the string `unknown` is returned.

Please read "INSPECTION" in **guestfs**(3) for more details. See also `$g->inspect_get_product_name`, `$g->inspect_get_major_version`.

`@roots` = `$g`−>inspect_get_roots ();

This function is a convenient way to get the list of root devices, as returned from a previous call to `$g->inspect_os`, but without redoing the whole inspection process.

This returns an empty list if either no root devices were found or the caller has not called `$g->inspect_os`.

Please read ''INSPECTION'' in **guestfs** (3) for more details.

`$name` = `$g`−>inspect_get_type ($root);

This returns the type of the inspected operating system. Currently defined types are:

''linux''
: Any Linux-based operating system.

''windows''
: Any Microsoft Windows operating system.

''freebsd''
: FreeBSD.

''netbsd''
: NetBSD.

''openbsd''
: OpenBSD.

''hurd''
: GNU/Hurd.

''dos''
: MS-DOS, FreeDOS and others.

''minix''
: MINIX.

''unknown''
: The operating system type could not be determined.

Future versions of libguestfs may return other strings here. The caller should be prepared to handle any string.

Please read ''INSPECTION'' in **guestfs** (3) for more details.

`$controlset` = `$g`−>inspect_get_windows_current_control_set ($root);

This returns the Windows CurrentControlSet of the inspected guest. The CurrentControlSet is a registry key name such as `ControlSet001`.

This call assumes that the guest is Windows and that the Registry could be examined by inspection. If this is not the case then an error is returned.

Please read ''INSPECTION'' in **guestfs** (3) for more details.

`$path` = `$g`−>inspect_get_windows_software_hive ($root);

This returns the path to the hive (binary Windows Registry file) corresponding to HKLM\SOFTWARE.

This call assumes that the guest is Windows and that the guest has a software hive file with the right name. If this is not the case then an error is returned. This call does not check that the hive is a valid Windows Registry hive.

You can use `$g->hivex_open` to read or write to the hive.

Please read ''INSPECTION'' in **guestfs** (3) for more details.

`$path` = `$g`−>inspect_get_windows_system_hive ($root);

This returns the path to the hive (binary Windows Registry file) corresponding to HKLM\SYSTEM.

This call assumes that the guest is Windows and that the guest has a system hive file with the right name. If this is not the case then an error is returned. This call does not check that the hive is a valid Windows Registry hive.

You can use `$g->hivex_open` to read or write to the hive.

Please read ‘‘INSPECTION’’ in **guestfs** (3) for more details.

`$systemroot` = `$g`−>inspect_get_windows_systemroot ($root);

This returns the Windows systemroot of the inspected guest. The systemroot is a directory path such as */WINDOWS*.

This call assumes that the guest is Windows and that the systemroot could be determined by inspection. If this is not the case then an error is returned.

Please read ‘‘INSPECTION’’ in **guestfs** (3) for more details.

`$live` = `$g`−>inspect_is_live ($root);

This is deprecated and always returns `false`.

Please read ‘‘INSPECTION’’ in **guestfs** (3) for more details.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs** (3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$multipart` = `$g`−>inspect_is_multipart ($root);

This is deprecated and always returns `false`.

Please read ‘‘INSPECTION’’ in **guestfs** (3) for more details.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs** (3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$netinst` = `$g`−>inspect_is_netinst ($root);

This is deprecated and always returns `false`.

Please read ‘‘INSPECTION’’ in **guestfs** (3) for more details.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs** (3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@applications` = `$g`−>inspect_list_applications ($root);

Return the list of applications installed in the operating system.

Note: This call works differently from other parts of the inspection API. You have to call `$g->inspect_os`, then `$g->inspect_get_mountpoints`, then mount up the disks, before calling this. Listing applications is a significantly more difficult operation which requires access to the full filesystem. Also note that unlike the other `$g->inspect_get_*` calls which are just returning data cached in the libguestfs handle, this call actually reads parts of the mounted filesystems during the call.

This returns an empty list if the inspection code was not able to determine the list of applications.

The application structure contains the following fields:

`app_name`
The name of the application. For Linux guests, this is the package name.

`app_display_name`
The display name of the application, sometimes localized to the install language of the guest operating system.

If unavailable this is returned as an empty string `""`. Callers needing to display something can use `app_name` instead.

`app_epoch`
For package managers which use epochs, this contains the epoch of the package (an integer). If unavailable, this is returned as `0`.

`app_version`
The version string of the application or package. If unavailable this is returned as an empty string `""`.

`app_release`
The release string of the application or package, for package managers that use this. If unavailable this is returned as an empty string `""`.

`app_install_path`
The installation path of the application (on operating systems such as Windows which use installation paths). This path is in the format used by the guest operating system, it is not a libguestfs path.

If unavailable this is returned as an empty string `""`.

`app_trans_path`
The install path translated into a libguestfs path. If unavailable this is returned as an empty string `""`.

`app_publisher`
The name of the publisher of the application, for package managers that use this. If unavailable this is returned as an empty string `""`.

`app_url`
The URL (eg. upstream URL) of the application. If unavailable this is returned as an empty string `""`.

`app_source_package`
For packaging systems which support this, the name of the source package. If unavailable this is returned as an empty string `""`.

`app_summary`
A short (usually one line) description of the application or package. If unavailable this is returned as an empty string `""`.

`app_description`
A longer description of the application or package. If unavailable this is returned as an empty string `""`.

Please read "INSPECTION" in **guestfs** (3) for more details.

This function is deprecated. In new code, use the "inspect_list_applications2" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@applications2` = `$g`->inspect_list_applications2 ($root);
Return the list of applications installed in the operating system.

Note: This call works differently from other parts of the inspection API. You have to call `$g->inspect_os`, then `$g->inspect_get_mountpoints`, then mount up the disks, before calling this. Listing applications is a significantly more difficult operation which requires access to the full filesystem. Also note that unlike the other `$g->inspect_get_*` calls which are just returning data cached in the libguestfs handle, this call actually reads parts of the mounted filesystems during the call.

This returns an empty list if the inspection code was not able to determine the list of applications.

The application structure contains the following fields:

`app2_name`
: The name of the application. For Linux guests, this is the package name.

`app2_display_name`
: The display name of the application, sometimes localized to the install language of the guest operating system.

  If unavailable this is returned as an empty string `""`. Callers needing to display something can use `app2_name` instead.

`app2_epoch`
: For package managers which use epochs, this contains the epoch of the package (an integer). If unavailable, this is returned as `0`.

`app2_version`
: The version string of the application or package. If unavailable this is returned as an empty string `""`.

`app2_release`
: The release string of the application or package, for package managers that use this. If unavailable this is returned as an empty string `""`.

`app2_arch`
: The architecture string of the application or package, for package managers that use this. If unavailable this is returned as an empty string `""`.

`app2_install_path`
: The installation path of the application (on operating systems such as Windows which use installation paths). This path is in the format used by the guest operating system, it is not a libguestfs path.

  If unavailable this is returned as an empty string `""`.

`app2_trans_path`
: The install path translated into a libguestfs path. If unavailable this is returned as an empty string `""`.

`app2_publisher`
: The name of the publisher of the application, for package managers that use this. If unavailable this is returned as an empty string `""`.

`app2_url`
: The URL (eg. upstream URL) of the application. If unavailable this is returned as an empty string `""`.

`app2_source_package`
: For packaging systems which support this, the name of the source package. If unavailable this is returned as an empty string `""`.

`app2_summary`
: A short (usually one line) description of the application or package. If unavailable this is returned as an empty string `""`.

`app2_description`
A longer description of the application or package. If unavailable this is returned as an empty string `""`.

Please read "INSPECTION" in **guestfs** (3) for more details.

`@roots` = `$g`−>inspect_os ();
This function uses other libguestfs functions and certain heuristics to inspect the disk(s) (usually disks belonging to a virtual machine), looking for operating systems.

The list returned is empty if no operating systems were found.

If one operating system was found, then this returns a list with a single element, which is the name of the root filesystem of this operating system. It is also possible for this function to return a list containing more than one element, indicating a dual-boot or multi-boot virtual machine, with each element being the root filesystem of one of the operating systems.

You can pass the root string(s) returned to other `$g->inspect_get_*` functions in order to query further information about each operating system, such as the name and version.

This function uses other libguestfs features such as `$g->mount_ro` and `$g->umount_all` in order to mount and unmount filesystems and look at the contents. This should be called with no disks currently mounted. The function may also use Augeas, so any existing Augeas handle will be closed.

This function cannot decrypt encrypted disks. The caller must do that first (supplying the necessary keys) if the disk is encrypted.

Please read "INSPECTION" in **guestfs** (3) for more details.

See also `$g->list_filesystems`.

`$flag` = `$g`−>is_blockdev ($path [, followsymlinks => `$followsymlinks`]);
This returns `true` if and only if there is a block device with the given `path` name.

If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a block device also causes the function to return true.

This call only looks at files within the guest filesystem. Libguestfs partitions and block devices (eg. */dev/sda*) cannot be used as the `path` parameter of this call.

See also `$g->stat`.

`$flag` = `$g`−>is_blockdev_opts ($path [, followsymlinks => `$followsymlinks`]);
This is an alias of "is_blockdev".

`$busy` = `$g`−>is_busy ();
This always returns false. This function is deprecated with no replacement. Do not use this function.

For more information on states, see **guestfs** (3).

`$flag` = `$g`−>is_chardev ($path [, followsymlinks => `$followsymlinks`]);
This returns `true` if and only if there is a character device with the given `path` name.

If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a chardev also causes the function to return true.

See also `$g->stat`.

`$flag` = `$g`−>is_chardev_opts ($path [, followsymlinks => `$followsymlinks`]);
This is an alias of "is_chardev".

`$config` = `$g`−>is_config ();
This returns true iff this handle is being configured (in the `CONFIG` state).

For more information on states, see **guestfs** (3).

`$dirflag` = `$g`−>is_dir ($path [, followsymlinks => `$followsymlinks`]);
: This returns `true` if and only if there is a directory with the given `path` name. Note that it returns false for other objects like files.

  If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a directory also causes the function to return true.

  See also `$g->stat`.

`$dirflag` = `$g`−>is_dir_opts ($path [, followsymlinks => `$followsymlinks`]);
: This is an alias of ‘‘is_dir’’.

`$flag` = `$g`−>is_fifo ($path [, followsymlinks => `$followsymlinks`]);
: This returns `true` if and only if there is a FIFO (named pipe) with the given `path` name.

  If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a FIFO also causes the function to return true.

  See also `$g->stat`.

`$flag` = `$g`−>is_fifo_opts ($path [, followsymlinks => `$followsymlinks`]);
: This is an alias of ‘‘is_fifo’’.

`$fileflag` = `$g`−>is_file ($path [, followsymlinks => `$followsymlinks`]);
: This returns `true` if and only if there is a regular file with the given `path` name. Note that it returns false for other objects like directories.

  If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a file also causes the function to return true.

  See also `$g->stat`.

`$fileflag` = `$g`−>is_file_opts ($path [, followsymlinks => `$followsymlinks`]);
: This is an alias of ‘‘is_file’’.

`$launching` = `$g`−>is_launching ();
: This returns true iff this handle is launching the subprocess (in the `LAUNCHING` state).

  For more information on states, see **guestfs** (3).

`$lvflag` = `$g`−>is_lv ($mountable);
: This command tests whether `mountable` is a logical volume, and returns true iff this is the case.

`$ready` = `$g`−>is_ready ();
: This returns true iff this handle is ready to accept commands (in the `READY` state).

  For more information on states, see **guestfs** (3).

`$flag` = `$g`−>is_socket ($path [, followsymlinks => `$followsymlinks`]);
: This returns `true` if and only if there is a Unix domain socket with the given `path` name.

  If the optional flag `followsymlinks` is true, then a symlink (or chain of symlinks) that ends with a socket also causes the function to return true.

  See also `$g->stat`.

`$flag` = `$g`−>is_socket_opts ($path [, followsymlinks => `$followsymlinks`]);
: This is an alias of ‘‘is_socket’’.

`$flag` = `$g`−>is_symlink ($path);
: This returns `true` if and only if there is a symbolic link with the given `path` name.

  See also `$g->stat`.

`$flag` = `$g`−>is_whole_device ($device);
: This returns `true` if and only if `device` refers to a whole block device. That is, not a partition or a logical device.

`$zeroflag` = `$g`−>is_zero ($path);
: This returns true iff the file exists and the file is empty or it contains all zero bytes.

`$zeroflag` = `$g`−>is_zero_device ($device);
: This returns true iff the device exists and contains all zero bytes.

  Note that for large devices this can take a long time to run.

`%isodata` = `$g`−>isoinfo ($isofile);
: This is the same as `$g->isoinfo_device` except that it works for an ISO file located inside some other mounted filesystem. Note that in the common case where you have added an ISO file as a libguestfs device, you would *not* call this. Instead you would call `$g->isoinfo_device`.

`%isodata` = `$g`−>isoinfo_device ($device);
: `device` is an ISO device. This returns a struct of information read from the primary volume descriptor (the ISO equivalent of the superblock) of the device.

  Usually it is more efficient to use the **isoinfo** (1) command with the *−d* option on the host to analyze ISO files, instead of going through libguestfs.

  For information on the primary volume descriptor fields, see <https://wiki.osdev.org/ISO_9660#The_Primary_Volume_Descriptor>

`$g`−>journal_close ();
: Close the journal handle.

  This function depends on the feature `journal`. See also`$g−>feature−available`.

`@fields` = `$g`−>journal_get ();
: Read the current journal entry. This returns all the fields in the journal as a set of`(attrname, attrval)` pairs. The `attrname` is the field name (a string).

  The `attrval` is the field value (a binary blob, often but not always a string). Please note that `attrval` is a byte array, *not* a \0−terminated C string.

  The length of data may be truncated to the data threshold (see: `$g->journal_set_data_threshold`, `$g−>journal_get_data_threshold`).

  If you set the data threshold to unlimited (`0`) then this call can read a journal entry of any size, ie. it is not limited by the libguestfs protocol.

  This function depends on the feature `journal`. See also`$g−>feature−available`.

`$threshold` = `$g`−>journal_get_data_threshold ();
: Get the current data threshold for reading journal entries. This is a hint to the journal that it may truncate data fields to this size when reading them (note also that it may not truncate them). If this returns `0`, then the threshold is unlimited.

  See also `$g−>journal_set_data_threshold`.

  This function depends on the feature `journal`. See also`$g−>feature−available`.

`$usec` = `$g`−>journal_get_realtime_usec ();
: Get the realtime (wallclock) timestamp of the current journal entry.

  This function depends on the feature `journal`. See also`$g−>feature−available`.

`$more` = `$g`−>journal_next ();
: Move to the next journal entry. You have to call this at least once after opening the handle before you are able to read data.

  The returned boolean tells you if there are any more journal records to read. `true` means you can read the next record (eg. using `$g−>journal_get`), and `false` means you have reached the end of the journal.

  This function depends on the feature `journal`. See also`$g−>feature−available`.

$g−>journal_open ($directory);

Open the systemd journal located in *directory*. Any previously opened journal handle is closed.

The contents of the journal can be read using `$g−>journal_next` and `$g−>journal_get`.

After you have finished using the journal, you should close the handle by calling `$g−>journal_close`.

This function depends on the feature `journal`. See also`$g−>feature−available`.

$g−>journal_set_data_threshold ($threshold);

Set the data threshold for reading journal entries. This is a hint to the journal that it may truncate data fields to this size when reading them (note also that it may not truncate them). If you set this to `0`, then the threshold is unlimited.

See also `$g−>journal_get_data_threshold`.

This function depends on the feature `journal`. See also`$g−>feature−available`.

$rskip = $g−>journal_skip ($skip);

Skip forwards (`skip X 0`) or backwards (`skip < 0`) in the journal.

The number of entries actually skipped is returned (note `rskip X 0`). If this is not the same as the absolute value of the skip parameter (`|skip|`) you passed in then it means you have reached the end or the start of the journal.

This function depends on the feature `journal`. See also`$g−>feature−available`.

$g−>kill_subprocess ();

This kills the hypervisor.

Do not call this. See: `$g−>shutdown` instead.

This function is deprecated. In new code, use the ''shutdown'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>launch ();

You should call this after configuring the handle (eg. adding drives) but before performing any actions.

Do not call `$g−>launch` twice on the same handle. Although it will not give an error (for historical reasons), the precise behaviour when you do this is not well defined. Handles are very cheap to create, so create a new one for each launch.

$g−>lchown ($owner, $group, $path);

Change the file owner to `owner` and group to `group`. This is like `$g−>chown` but if `path` is a symlink then the link itself is changed, not the target.

Only numeric uid and gid are supported. If you want to use names, you will need to locate and parse the password file yourself (Augeas support makes this relatively easy).

$g−>ldmtool_create_all ();

This function scans all block devices looking for Windows dynamic disk volumes and partitions, and creates devices for any that were found.

Call `$g−>list_ldm_volumes` and `$g−>list_ldm_partitions` to return all devices.

Note that you **don't** normally need to call this explicitly, since it is done automatically at `$g−>launch` time. However you might want to call this function if you have hotplugged disks or have just created a Windows dynamic disk.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`@disks` = `$g`−>ldmtool_diskgroup_disks ($diskgroup);

Return the disks in a Windows dynamic disk group. The `diskgroup` parameter should be the GUID of a disk group, one element from the list returned by `$g−>ldmtool_scan`.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`$name` = `$g`−>ldmtool_diskgroup_name ($diskgroup);

Return the name of a Windows dynamic disk group. The `diskgroup` parameter should be the GUID of a disk group, one element from the list returned by `$g−>ldmtool_scan`.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`@volumes` = `$g`−>ldmtool_diskgroup_volumes ($diskgroup);

Return the volumes in a Windows dynamic disk group. The `diskgroup` parameter should be the GUID of a disk group, one element from the list returned by `$g−>ldmtool_scan`.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`$g`−>ldmtool_remove_all ();

This is essentially the opposite of `$g−>ldmtool_create_all`. It removes the device mapper mappings for all Windows dynamic disk volumes

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`@guids` = `$g`−>ldmtool_scan ();

This function scans for Windows dynamic disks. It returns a list of identifiers (GUIDs) for all disk groups that were found. These identifiers can be passed to other `$g−>ldmtool_*` functions.

This function scans all block devices. To scan a subset of block devices, call `$g−>ldmtool_scan_devices` instead.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`@guids` = `$g`−>ldmtool_scan_devices (\@devices);

This function scans for Windows dynamic disks. It returns a list of identifiers (GUIDs) for all disk groups that were found. These identifiers can be passed to other `$g−>ldmtool_*` functions.

The parameter `devices` is a list of block devices which are scanned. If this list is empty, all block devices are scanned.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`$hint` = `$g`−>ldmtool_volume_hint ($diskgroup, `$volume`);

Return the hint field of the volume named `volume` in the disk group with GUID `diskgroup`. This may not be defined, in which case the empty string is returned. The hint field is often, though not always, the name of a Windows drive, eg. `E:`.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`@partitions` = `$g`−>ldmtool_volume_partitions ($diskgroup, `$volume`);

Return the list of partitions in the volume named `volume` in the disk group with GUID `diskgroup`.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`$voltype` = `$g`−>ldmtool_volume_type ($diskgroup, `$volume`);

Return the type of the volume named `volume` in the disk group with GUID `diskgroup`.

Possible volume types that can be returned here include: `simple`, `spanned`, `striped`, `mirrored`, `raid5`. Other types may also be returned.

This function depends on the feature `ldm`. See also`$g−>feature−available`.

`$xattr` = `$g`−>lgetxattr ($path, `$name`);

Get a single extended attribute from file `path` named `name`. If`path` is a symlink, then this call returns an extended attribute from the symlink.

Normally it is better to get all extended attributes from a file in one go by calling `$g->getxattrs`. However some Linux filesystem implementations are buggy and do not provide a way to list out attributes. For these filesystems (notably ntfs−3g) you have to know the names of the extended attributes you want in advance and call this function.

Extended attribute values are blobs of binary data. If there is no extended attribute named `name`, this returns an error.

See also: `$g->lgetxattrs`, `$g->getxattr`, **attr** (5).

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`@xattrs` = `$g`−>lgetxattrs ($path);

This is the same as `$g->getxattrs`, but if `path` is a symbolic link, then it returns the extended attributes of the link itself.

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`@mounttags` = `$g`−>list_9p ();

List all 9p filesystems attached to the guest. A list of mount tags is returned.

`@devices` = `$g`−>list_devices ();

List all the block devices.

The full block device names are returned, eg. */dev/sda*.

See also `$g->list_filesystems`.

`%labels` = `$g`−>list_disk_labels ();

If you add drives using the optional `label` parameter of `$g->add_drive_opts`, you can use this call to map between disk labels, and raw block device and partition names (like */dev/sda* and */dev/sda1*).

This returns a hashtable, where keys are the disk labels (*without* the */dev/disk/guestfs* prefix), and the values are the full raw block device and partition names (eg. */dev/sda* and */dev/sda1*).

`@devices` = `$g`−>list_dm_devices ();

List all device mapper devices.

The returned list contains */dev/mapper/** devices, eg. ones created by a previous call to `$g->luks_open`.

Device mapper devices which correspond to logical volumes are *not* returned in this list. Call `$g->lvs` if you want to list logical volumes.

`%fses` = `$g`−>list_filesystems ();

This inspection command looks for filesystems on partitions, block devices and logical volumes, returning a list of `mountables` containing filesystems and their type.

The return value is a hash, where the keys are the devices containing filesystems, and the values are the filesystem types. For example:

```
 "/dev/sda1" => "ntfs"
 "/dev/sda2" => "ext2"
 "/dev/vg_guest/lv_root" => "ext4"
 "/dev/vg_guest/lv_swap" => "swap"
```

The key is not necessarily a block device. It may also be an opaque XmountableX string which can be passed to `$g->mount`.

The value can have the special value ''unknown'', meaning the content of the device is undetermined or empty. ''swap'' means a Linux swap partition.

In libguestfs X 1.36 this command ran other libguestfs commands, which might have included `$g->mount` and `$g->umount`, and therefore you had to use this soon after launch and only when nothing else was mounted. This restriction is removed in libguestfs X 1.38.

Not all of the filesystems returned will be mountable. In particular, swap partitions are returned in the list. Also this command does not check that each filesystem found is valid and mountable, and some filesystems might be mountable but require special options. Filesystems may not all belong to a single logical operating system (use `$g->inspect_os` to look for OSes).

`@devices` = `$g`−>list_ldm_partitions ();
This function returns all Windows dynamic disk partitions that were found at launch time. It returns a list of device names.

This function depends on the feature `ldm`. See also`$g->feature-available`.

`@devices` = `$g`−>list_ldm_volumes ();
This function returns all Windows dynamic disk volumes that were found at launch time. It returns a list of device names.

This function depends on the feature `ldm`. See also`$g->feature-available`.

`@devices` = `$g`−>list_md_devices ();
List all Linux md devices.

`@partitions` = `$g`−>list_partitions ();
List all the partitions detected on all block devices.

The full partition device names are returned, eg. */dev/sda1*

This does not return logical volumes. For that you will need to call `$g->lvs`.

See also `$g->list_filesystems`.

`$listing` = `$g`−>ll ($directory);
List the files in *directory* (relative to the root directory, there is no cwd) in the format of `ls -la`.

This command is mostly useful for interactive sessions. It is *not* intended that you try to parse the output string.

`$listing` = `$g`−>llz ($directory);
List the files in *directory* in the format of `ls -laZ`.

This command is mostly useful for interactive sessions. It is *not* intended that you try to parse the output string.

This function is deprecated. In new code, use the ''lgetxattrs'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>ln ($target, `$linkname`);
This command creates a hard link.

`$g`−>ln_f ($target, `$linkname`);
This command creates a hard link, removing the link `linkname` if it exists already.

`$g`−>ln_s ($target, `$linkname`);
This command creates a symbolic link using the `ln -s` command.

`$g`−>ln_sf ($target, `$linkname`);
This command creates a symbolic link using the `ln -sf` command, The *−f* option removes the link (`linkname`) if it exists already.

`$g`−>lremovexattr ($xattr, `$path`);
This is the same as `$g->removexattr`, but if `path` is a symbolic link, then it removes an extended attribute of the link itself.

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`@listing` = `$g`−>ls ($directory);

List the files in *directory* (relative to the root directory, there is no cwd). The `.` and `..` entries are not returned, but hidden files are shown.

`$g`−>ls0 ($dir, `$filenames`);

This specialized command is used to get a listing of the filenames in the directory `dir`. The list of filenames is written to the local file *filenames* (on the host).

In the output file, the filenames are separated by `\0` characters.

`.` and `..` are not returned. The filenames are not sorted.

`$g`−>lsetxattr ($xattr, `$val`, `$vallen`, `$path`);

This is the same as `$g->setxattr`, but if `path` is a symbolic link, then it sets an extended attribute of the link itself.

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`%statbuf` = `$g`−>lstat ($path);

Returns file information for the given `path`.

This is the same as `$g->stat` except that if `path` is a symbolic link, then the link is stat-ed, not the file it refers to.

This is the same as the **lstat** (2) system call.

This function is deprecated. In new code, use the "lstatns" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@statbufs` = `$g`−>lstatlist ($path, \@names);

This call allows you to perform the `$g->lstat` operation on multiple files, where all files are in the directory `path`. `names` is the list of files from this directory.

On return you get a list of stat structs, with a one-to-one correspondence to the `names` list. If any name did not exist or could not be lstat'd, then the `st_ino` field of that structure is set to `-1`.

This call is intended for programs that want to efficiently list a directory contents without making many round-trips. See also `$g->lxattrlist` for a similarly efficient call for getting extended attributes.

This function is deprecated. In new code, use the "lstatnslist" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`%statbuf` = `$g`−>lstatns ($path);

Returns file information for the given `path`.

This is the same as `$g->statns` except that if `path` is a symbolic link, then the link is stat-ed, not the file it refers to.

This is the same as the **lstat** (2) system call.

`@statbufs` = `$g`−>lstatnslist ($path, \@names);

This call allows you to perform the `$g->lstatns` operation on multiple files, where all files are in the directory `path`. `names` is the list of files from this directory.

On return you get a list of stat structs, with a one-to-one correspondence to the `names` list. If any name did not exist or could not be lstat'd, then the `st_ino` field of that structure is set to `-1`.

This call is intended for programs that want to efficiently list a directory contents without making many round-trips. See also `$g->lxattrlist` for a similarly efficient call for getting extended attributes.

$g−>luks_add_key ($device, $key, $newkey, $keyslot);
: This command adds a new key on LUKS device `device`. `key` is any existing key, and is used to access the device. `newkey` is the new key to add. `keyslot` is the key slot that will be replaced.

  Note that if `keyslot` already contains a key, then this command will fail. You have to use `$g−>luks_kill_slot` first to remove that key.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

$g−>luks_close ($device);
: This closes a LUKS device that was created earlier by `$g−>luks_open` or `$g−>luks_open_ro`. The `device` parameter must be the name of the LUKS mapping device (ie. */dev/mapper/mapname*) and *not* the name of the underlying block device.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

  This function is deprecated. In new code, use the ''cryptsetup_close'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>luks_format ($device, $key, $keyslot);
: This command erases existing data on `device` and formats the device as a LUKS encrypted device. `key` is the initial key, which is added to key slot `keyslot`. (LUKS supports 8 key slots, numbered 0−7).

  This function depends on the feature `luks`. See also`$g−>feature−available`.

$g−>luks_format_cipher ($device, $key, $keyslot, $cipher);
: This command is the same as `$g−>luks_format` but it also allows you to set the `cipher` used.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

$g−>luks_kill_slot ($device, $key, $keyslot);
: This command deletes the key in key slot `keyslot` from the encrypted LUKS device `device`. `key` must be one of the *other* keys.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

$g−>luks_open ($device, $key, $mapname);
: This command opens a block device which has been encrypted according to the Linux Unified Key Setup (LUKS) standard.

  `device` is the encrypted block device or partition.

  The caller must supply one of the keys associated with the LUKS block device, in the `key` parameter.

  This creates a new block device called */dev/mapper/mapname*. Reads and writes to this block device are decrypted from and encrypted to the underlying `device` respectively.

  If this block device contains LVM volume groups, then calling `$g−>lvm_scan` with the `activate` parameter `true` will make them visible.

  Use `$g−>list_dm_devices` to list all device mapper devices.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

  This function is deprecated. In new code, use the ''cryptsetup_open'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>luks_open_ro ($device, $key, $mapname);
: This is the same as `$g−>luks_open` except that a read-only mapping is created.

  This function depends on the feature `luks`. See also`$g−>feature−available`.

This function is deprecated. In new code, use the "cryptsetup_open" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$uuid` = `$g`−>luks_uuid ($device);

This returns the UUID of the LUKS device `device`.

This function depends on the feature `luks`. See also`$g->feature−available`.

`$g`−>lvcreate ($logvol, `$volgroup`, `$mbytes`);

This creates an LVM logical volume called `logvol` on the volume group `volgroup`, with `size` megabytes.

This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>lvcreate_free ($logvol, `$volgroup`, `$percent`);

Create an LVM logical volume called */dev/volgroup/logvol*, using approximately `percent` % of the free space remaining in the volume group. Most usefully, when `percent` is `100` this will create the largest possible LV.

This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$lv` = `$g`−>lvm_canonical_lv_name ($lvname);

This converts alternative naming schemes for LVs that you might find to the canonical name. For example, */dev/mapper/VG−LV* is converted to */dev/VG/LV*.

This command returns an error if the `lvname` parameter does not refer to a logical volume. In this case errno will be set to `EINVAL`.

See also `$g->is_lv`, `$g->canonical_device_name`.

`$g`−>lvm_clear_filter ();

This undoes the effect of `$g->lvm_set_filter`. LVM will be able to see every block device.

This command also clears the LVM cache and performs a volume group scan.

`$g`−>lvm_remove_all ();

This command removes all LVM logical volumes, volume groups and physical volumes.

This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>lvm_scan ($activate);

This scans all block devices and rebuilds the list of LVM physical volumes, volume groups and logical volumes.

If the `activate` parameter is `true` then newly found volume groups and logical volumes are activated, meaning the LV */dev/VG/LV* devices become visible.

When a libguestfs handle is launched it scans for existing devices, so you do not normally need to use this API. However it is useful when you have added a new device or deleted an existing device (such as when the `$g->luks_open` API is used).

`$g`−>lvm_set_filter (\@devices);

This sets the LVM device filter so that LVM will only be able to "see" the block devices in the list `devices`, and will ignore all other attached block devices.

Where disk image(s) contain duplicate PVs or VGs, this command is useful to get LVM to ignore the duplicates, otherwise LVM can get confused. Note also there are two types of duplication possible: either cloned PVs/VGs which have identical UUIDs; or VGs that are not cloned but just happen to have the same name. In normal operation you cannot create this situation, but you can do it outside LVM, eg. by cloning disk images or by bit twiddling inside the LVM metadata.

This command also clears the LVM cache and performs a volume group scan.

You can filter whole block devices or individual partitions.

You cannot use this if any VG is currently in use (eg. contains a mounted filesystem), even if you are not filtering out that VG.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`->lvremove ($device);
Remove an LVM logical volume `device`, where `device` is the path to the LV, such as */dev/VG/LV*.

You can also remove all LVs in a volume group by specifying the VG name, */dev/VG*.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`->lvrename ($logvol, `$newlogvol`);
Rename a logical volume `logvol` with the new name `newlogvol`.

`$g`->lvresize ($device, `$mbytes`);
This resizes (expands or shrinks) an existing LVM logical volume to `mbytes`. When reducing, data in the reduced part is lost.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`->lvresize_free ($lv, `$percent`);
This expands an existing logical volume `lv` so that it fills `pc` % of the remaining free space in the volume group. Commonly you would call this with pc = 100 which expands the logical volume as much as possible, using all remaining free space in the volume group.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`@logvols` = `$g`->lvs ();
List all the logical volumes detected. This is the equivalent of the **lvs**(8) command.

This returns a list of the logical volume device names (eg. */dev/VolGroup00/LogVol00*).

See also `$g->lvs_full`, `$g->list_filesystems`.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`@logvols` = `$g`->lvs_full ();
List all the logical volumes detected. This is the equivalent of the **lvs**(8) command. The ‘‘full’’ version includes all fields.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$uuid` = `$g`->lvuuid ($device);
This command returns the UUID of the LVM LV `device`.

`@xattrs` = `$g`->lxattrlist ($path, \@names);
This call allows you to get the extended attributes of multiple files, where all files are in the directory `path`. `names` is the list of files from this directory.

On return you get a flat list of xattr structs which must be interpreted sequentially. The first xattr struct always has a zero-length `attrname`. `attrval` in this struct is zero-length to indicate there was an error doing `$g->lgetxattr` for this file, *or* is a C string which is a decimal number (the number of following attributes for this file, which could be `"0"`). Then after the first xattr struct are the zero or more attributes for the first named file. This repeats for the second and subsequent files.

This call is intended for programs that want to efficiently list a directory contents without making many round-trips. See also `$g->lstatlist` for a similarly efficient call for getting standard stats.

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`$disks` = `$g`->max_disks ();
Return the maximum number of disks that may be added to a handle (eg. by `$g->add_drive_opts` and similar calls).

This function was added in libguestfs 1.19.7. In previous versions of libguestfs the limit was 25.

See "MAXIMUM NUMBER OF DISKS" in **guestfs** (3) for additional information on this topic.

`$g`−>md_create ($name, \@devices [, missingbitmap => `$missingbitmap`] [, nrdevices => `$nrdevices`] [, spare => `$spare`] [, chunk => `$chunk`] [, level => `$level`]);

Create a Linux md (RAID) device named `name` on the devices in the list `devices`.

The optional parameters are:

`missingbitmap`

A bitmap of missing devices. If a bit is set it means that a missing device is added to the array. The least significant bit corresponds to the first device in the array.

As examples:

If `devices = ["/dev/sda"]` and `missingbitmap = 0x1` then the resulting array would be `[<missing>, "/dev/sda"]`.

If `devices = ["/dev/sda"]` and `missingbitmap = 0x2` then the resulting array would be `["/dev/sda", <missing>]`.

This defaults to `0` (no missing devices).

The length of `devices` + the number of bits set in `missingbitmap` must equal `nrdevices` + `spare`.

`nrdevices`

The number of active RAID devices.

If not set, this defaults to the length of `devices` plus the number of bits set in `missingbitmap`.

`spare`

The number of spare devices.

If not set, this defaults to `0`.

`chunk`

The chunk size in bytes.

`level`

The RAID level, which can be one of: `linear`, `raid0`, `0`, `stripe`, `raid1`, `1`, `mirror`, `raid4`, `4`, `raid5`, `5`, `raid6`, `6`, `raid10`, `10`. Some of these are synonymous, and more levels may be added in future.

If not set, this defaults to `raid1`.

This function depends on the feature `mdadm`. See also`$g−>feature−available`.

`%info` = `$g`−>md_detail ($md);

This command exposes the output of `mdadm −DY <md>`. The following fields are usually present in the returned hash. Other fields may also be present.

`level`

The raid level of the MD device.

`devices`

The number of underlying devices in the MD device.

`metadata`

The metadata version used.

`uuid`

The UUID of the MD device.

`name`

The name of the MD device.

This function depends on the feature `mdadm`. See also`$g->feature-available`.

`@devices` = `$g`->md_stat ($md);

This call returns a list of the underlying devices which make up the single software RAID array device `md`.

To get a list of software RAID devices, call `$g->list_md_devices`.

Each structure returned corresponds to one device along with additional status information:

`mdstat_device`

The name of the underlying device.

`mdstat_index`

The index of this device within the array.

`mdstat_flags`

Flags associated with this device. This is a string containing (in no specific order) zero or more of the following flags:

`W`	write-mostly
`F`	device is faulty
`S`	device is a RAID spare
`R`	replacement

This function depends on the feature `mdadm`. See also`$g->feature-available`.

`$g`->md_stop ($md);

This command deactivates the MD array named `md`. The device is stopped, but it is not destroyed or zeroed.

This function depends on the feature `mdadm`. See also`$g->feature-available`.

`$g`->mkdir ($path);

Create a directory named `path`.

`$g`->mkdir_mode ($path, `$mode`);

This command creates a directory, setting the initial permissions of the directory to `mode`.

For common Linux filesystems, the actual mode which is set will be `mode & ~umask & 01777`. Non-native-Linux filesystems may interpret the mode in other ways.

See also `$g->mkdir`, `$g->umask`

`$g`->mkdir_p ($path);

Create a directory named `path`, creating any parent directories as necessary. This is like the `mkdir -p` shell command.

`$dir` = `$g`->mkdtemp ($tmpl);

This command creates a temporary directory. The`tmpl` parameter should be a full pathname for the temporary directory name with the final six characters being ‘‘XXXXXX’’.

For example: ‘‘/tmp/myprogXXXXXX’’ or ‘‘/Temp/myprogXXXXXX’’, the second one being suitable for Windows filesystems.

The name of the temporary directory that was created is returned.

The temporary directory is created with mode 0700 and is owned by root.

The caller is responsible for deleting the temporary directory and its contents after use.

See also: **mkdtemp** (3)

$g−>mke2fs ($device [, blockscount => $blockscount] [, blocksize => $blocksize] [, fragsize => $fragsize] [, blockspergroup => $blockspergroup] [, numberofgroups => $numberofgroups] [, bytesperinode => $bytesperinode] [, inodesize => $inodesize] [, journalsize => $journalsize] [, numberofinodes => $numberofinodes] [, stridesize => $stridesize] [, stripewidth => $stripewidth] [, maxonlineresize => $maxonlineresize] [, reservedblockspercentage => $reservedblockspercentage] [, mmpupdateinterval => $mmpupdateinterval] [, journaldevice => $journaldevice] [, label => $label] [, lastmounteddir => $lastmounteddir] [, creatoros => $creatoros] [, fstype => $fstype] [, usagetype => $usagetype] [, uuid => $uuid] [, forcecreate => $forcecreate] [, writesbandgrouponly => $writesbandgrouponly] [, lazyitableinit => $lazyitableinit] [, lazyjournalinit => $lazyjournalinit] [, testfs => $testfs] [, discard => $discard] [, quotatype => $quotatype] [, extent => $extent] [, filetype => $filetype] [, flexbg => $flexbg] [, hasjournal => $hasjournal] [, journaldev => $journaldev] [, largefile => $largefile] [, quota => $quota] [, resizeinode => $resizeinode] [, sparsesuper => $sparsesuper] [, uninitbg => $uninitbg]);

mke2fs is used to create an ext2, ext3, or ext4 filesystem on device.

The optional blockscount is the size of the filesystem in blocks. If omitted it defaults to the size of device. Note if the filesystem is too small to contain a journal,mke2fs will silently create an e xt2 filesystem instead.

$g−>mke2fs_J ($fstype, $blocksize, $device, $journal);

This creates an ext2/3/4 filesystem on device with an external journal on journal. It is equivalent to the command:

```
 mke2fs −t fstype −b blocksize −J device=<journal> <device>
```

See also $g−>mke2journal.

This function is deprecated. In new code, use the ‘‘mke2fs’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>mke2fs_JL ($fstype, $blocksize, $device, $label);

This creates an ext2/3/4 filesystem on device with an external journal on the journal labeled label.

See also $g−>mke2journal_L.

This function is deprecated. In new code, use the ‘‘mke2fs’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>mke2fs_JU ($fstype, $blocksize, $device, $uuid);

This creates an ext2/3/4 filesystem on device with an external journal on the journal with UUID uuid.

See also $g−>mke2journal_U.

This function depends on the feature linuxfsuuid. See also$g−>feature−available.

This function is deprecated. In new code, use the ‘‘mke2fs’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>mke2journal ($blocksize, $device);

This creates an ext2 external journal on device. It is equivalent to the command:

```
 mke2fs −O journal_dev −b blocksize device
```

This function is deprecated. In new code, use the ‘‘mke2fs’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>mke2journal_L ($blocksize, `$label`, `$device`);

This creates an ext2 external journal on `device` with label `label`.

This function is deprecated. In new code, use the ''mke2fs'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>mke2journal_U ($blocksize, `$uuid`, `$device`);

This creates an ext2 external journal on `device` with UUID `uuid`.

This function depends on the feature `linuxfsuuid`. See also`$g->feature-available`.

This function is deprecated. In new code, use the ''mke2fs'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>mkfifo ($mode, `$path`);

This call creates a FIFO (named pipe) called `path` with mode `mode`. It is just a convenient wrapper around `$g->mknod`.

Unlike with `$g->mknod`, `mode` **must** contain only permissions bits.

The mode actually set is affected by the umask.

This function depends on the feature `mknod`. See also`$g->feature-available`.

`$g`−>mkfs ($fstype, `$device` [, blocksize => `$blocksize`] [, features => `$features`] [, inode => `$inode`] [, sectorsize => `$sectorsize`] [, label => `$label`]);

This function creates a filesystem on `device`. The filesystem type is`fstype`, for e xample `ext3`.

The optional arguments are:

`blocksize`

The filesystem block size. Supported block sizes depend on the filesystem type, but typically they are `1024`, `2048` or `4096` for Linux ext2/3 filesystems.

For VFAT and NTFS the `blocksize` parameter is treated as the requested cluster size.

For UFS block sizes, please see **mkfs.ufs** (8).

`features`

This passes the *−O* parameter to the external mkfs program.

For certain filesystem types, this allows extra filesystem features to be selected. See **mke2fs** (8) and **mkfs.ufs** (8) for more details.

You cannot use this optional parameter with the `gfs` or `gfs2` filesystem type.

`inode`

This passes the *−I* parameter to the external **mke2fs** (8) program which sets the inode size (only for ext2/3/4 filesystems at present).

`sectorsize`

This passes the *−S* parameter to external **mkfs.ufs** (8) program, which sets sector size for ufs filesystem.

`$g`−>mkfs_opts ($fstype, `$device` [, blocksize => `$blocksize`] [, features => `$features`] [, inode => `$inode`] [, sectorsize => `$sectorsize`] [, label => `$label`]);

This is an alias of ''mkfs''.

$g−>mkfs_b ($fstype, $blocksize, $device);

This call is similar to $g−>mkfs, but it allows you to control the block size of the resulting filesystem. Supported block sizes depend on the filesystem type, but typically they are 1024, 2048 or 4096 only.

For VFAT and NTFS the blocksize parameter is treated as the requested cluster size.

This function is deprecated. In new code, use the ''mkfs'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>mkfs_btrfs (\@devices [, allocstart => $allocstart] [, bytecount => $bytecount] [, datatype => $datatype] [, leafsize => $leafsize] [, label => $label] [, metadata => $metadata] [, nodesize => $nodesize] [, sectorsize => $sectorsize]);

Create a btrfs filesystem, allowing all configurables to be set. For more information on the optional arguments, see **mkfs.btrfs** (8).

Since btrfs filesystems can span multiple devices, this takes a non-empty list of devices.

To create general filesystems, use $g−>mkfs.

This function depends on the feature btrfs. See also$g−>feature−available.

$g−>mklost_and_found ($mountpoint);

Make the lost+found directory, normally in the root directory of an ext2/3/4 filesystem. mountpoint is the directory under which we try to create the lost+found directory.

$g−>mkmountpoint ($exemptpath);

$g−>mkmountpoint and $g−>rmmountpoint are specialized calls that can be used to create extra mountpoints before mounting the first filesystem.

These calls are *only* necessary in some very limited circumstances, mainly the case where you want to mount a mix of unrelated and/or read-only filesystems together.

For example, live CDs often contain a ''Russian doll'' nest of filesystems, an ISO outer layer, with a squashfs image inside, with an ext2/3 image inside that. You can unpack this as follows in guestfish:

```
add−ro Fedora−11−i686−Live.iso
run
mkmountpoint /cd
mkmountpoint /sqsh
mkmountpoint /ext3fs
mount /dev/sda /cd
mount−loop /cd/LiveOS/squashfs.img /sqsh
mount−loop /sqsh/LiveOS/ext3fs.img /ext3fs
```

The inner filesystem is now unpacked under the /ext3fs mountpoint.

$g−>mkmountpoint is not compatible with $g−>umount_all. You may get unexpected errors if you try to mix these calls. It is safest to manually unmount filesystems and remove mountpoints after use.

$g−>umount_all unmounts filesystems by sorting the paths longest first, so for this to work for manual mountpoints, you must ensure that the innermost mountpoints have the longest pathnames, as in the example code above.

For more details see <https://bugzilla.redhat.com/show_bug.cgi?id=599503>

Autosync [see $g−>set_autosync, this is set by default on handles] can cause $g−>umount_all to be called when the handle is closed which can also trigger these issues.

$g−>mknod ($mode, $devmajor, $devminor, $path);

This call creates block or character special devices, or named pipes (FIFOs).

The mode parameter should be the mode, using the standard constants. devmajor and devminor are the device major and minor numbers, only used when creating block and character special devices.

Note that, just like **mknod** (2), the mode must be bitwise OR'd with S_IFBLK, S_IFCHR, S_IFIFO or S_IFSOCK (otherwise this call just creates a regular file). These constants are available in the standard Linux header files, or you can use $g−>mknod_b, $g−>mknod_c or $g−>mkfifo which are wrappers around this command which bitwise OR in the appropriate constant for you.

The mode actually set is affected by the umask.

This function depends on the feature mknod. See also$g−>feature−available.

$g−>mknod_b ($mode, $devmajor, $devminor, $path);

This call creates a block device node called path with mode mode and device major/minor devmajor and devminor. It is just a convenient wrapper around $g−>mknod.

Unlike with $g−>mknod, mode **must** contain only permissions bits.

The mode actually set is affected by the umask.

This function depends on the feature mknod. See also$g−>feature−available.

$g−>mknod_c ($mode, $devmajor, $devminor, $path);

This call creates a char device node called path with mode mode and device major/minor devmajor and devminor. It is just a convenient wrapper around $g−>mknod.

Unlike with $g−>mknod, mode **must** contain only permissions bits.

The mode actually set is affected by the umask.

This function depends on the feature mknod. See also$g−>feature−available.

$g−>mksquashfs ($path, $filename [, compress => $compress] [, excludes => $excludes]);

Create a squashfs filesystem for the specified path.

The optional compress flag controls compression. If not given, then the output compressed using gzip. Otherwise one of the following strings may be given to select the compression type of the squashfs: gzip, lzma, lzo, lz4, xz.

The other optional arguments are:

excludes

A list of wildcards. Files are excluded if they match any of the wildcards.

Please note that this API may fail when used to compress directories with large files, such as the resulting squashfs will be over 3GB big.

This function depends on the feature squashfs. See also$g−>feature−available.

$g−>mkswap ($device [, label => $label] [, uuid => $uuid]);

Create a Linux swap partition on device.

The option arguments label and uuid allow you to set the label and/or UUID of the new swap partition.

$g−>mkswap_opts ($device [, label => $label] [, uuid => $uuid]);

This is an alias of "mkswap".

$g−>mkswap_L ($label, $device);

Create a swap partition on device with label label.

Note that you cannot attach a swap label to a block device (eg. */dev/sda*), just to a partition. This appears to be a limitation of the kernel or swap tools.

This function is deprecated. In new code, use the "mkswap" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>mkswap_U ($uuid, `$device`);

Create a swap partition on `device` with UUID `uuid`.

This function depends on the feature `linuxfsuuid`. See also`$g->feature-available`.

This function is deprecated. In new code, use the "mkswap" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>mkswap_file ($path);

Create a swap file.

This command just writes a swap file signature to an existing file. To create the file itself, use something like `$g->fallocate`.

`$path` = `$g`−>mktemp ($tmpl [, suffix => `$suffix`]);

This command creates a temporary file. The `tmpl` parameter should be a full pathname for the temporary directory name with the final six characters being "XXXXXX".

For example: "/tmp/myprogXXXXXX" or "/Temp/myprogXXXXXX", the second one being suitable for Windows filesystems.

The name of the temporary file that was created is returned.

The temporary file is created with mode 0600 and is owned by root.

The caller is responsible for deleting the temporary file after use.

If the optional `suffix` parameter is given, then the suffix (eg. `.txt`) is appended to the temporary name.

See also: `$g->mkdtemp`.

`$g`−>modprobe ($modulename);

This loads a kernel module in the appliance.

This function depends on the feature `linuxmodules`. See also`$g->feature-available`.

`$g`−>mount ($mountable, `$mountpoint`);

Mount a guest disk at a position in the filesystem. Block devices are named */dev/sda*, */dev/sdb* and so on, as they were added to the guest. If those block devices contain partitions, they will have the usual names (eg. */dev/sda1*). Also LVM */dev/VG/LV*−style names can be used, or XmountableX strings returned by `$g->list_filesystems` or `$g->inspect_get_mountpoints`.

The rules are the same as for **mount** (2): A filesystem must first be mounted on*/* before others can be mounted. Other filesystems can only be mounted on directories which already exist.

The mounted filesystem is writable, if we have sufficient permissions on the underlying device.

Before libguestfs 1.13.16, this call implicitly added the options `sync` and `noatime`. The`sync` option greatly slowed writes and caused many problems for users. If your program might need to work with older versions of libguestfs, use `$g->mount_options` instead (using an empty string for the first parameter if you don't want any options).

`$g`−>mount_9p ($mounttag, `$mountpoint` [, options => `$options`]);

Mount the virtio−9p filesystem with the tag `mounttag` on the directory `mountpoint`.

If required, `trans=virtio` will be automatically added to the options. Any other options required can be passed in the optional `options` parameter.

`$g`−>mount_local ($localmountpoint [, readonly => `$readonly`] [, options => `$options`] [, cachetimeout => `$cachetimeout`] [, debugcalls => `$debugcalls`]);

This call exports the libguestfs-accessible filesystem to a local mountpoint (directory) called `localmountpoint`. Ordinary reads and writes to files and directories under`localmountpoint` are redirected through libguestfs.

If the optional `readonly` flag is set to true, then writes to the filesystem return error `EROFS`.

`options` is a comma-separated list of mount options. See **guestmount** (1) for some useful options.

`cachetimeout` sets the timeout (in seconds) for cached directory entries. The default is 60 seconds. See**guestmount** (1) for further information.

If `debugcalls` is set to true, then additional debugging information is generated for every FUSE call.

When `$g−>mount_local` returns, the filesystem is ready, but is not processing requests (access to it will block). You have to call `$g−>mount_local_run` to run the main loop.

See "MOUNT LOCAL" in **guestfs** (3) for full documentation.

`$g`−>mount_local_run ();

Run the main loop which translates kernel calls to libguestfs calls.

This should only be called after `$g−>mount_local` returns successfully. The call will not return until the filesystem is unmounted.

Note you must *not* make concurrent libguestfs calls on the same handle from another thread.

You may call this from a different thread than the one which called `$g−>mount_local`, subject to the usual rules for threads and libguestfs (see "MULTIPLE HANDLES AND MULTIPLE THREADS" in **guestfs** (3)).

See "MOUNT LOCAL" in **guestfs** (3) for full documentation.

`$g`−>mount_loop ($file, `$mountpoint`);

This command lets you mount *file* (a filesystem image in a file) on a mount point. It is entirely equivalent to the command `mount −o loop file mountpoint`.

`$g`−>mount_options ($options, `$mountable`, `$mountpoint`);

This is the same as the `$g−>mount` command, but it allows you to set the mount options as for the **mount** (8) *−o* flag.

If the `options` parameter is an empty string, then no options are passed (all options default to whatever the filesystem uses).

`$g`−>mount_ro ($mountable, `$mountpoint`);

This is the same as the `$g−>mount` command, but it mounts the filesystem with the read-only (*−o ro*) flag.

`$g`−>mount_vfs ($options, `$vfstype`, `$mountable`, `$mountpoint`);

This is the same as the `$g−>mount` command, but it allows you to set both the mount options and the vfstype as for the **mount** (8) *−o* and *−t* flags.

`$device` = `$g`−>mountable_device ($mountable);

Returns the device name of a mountable. In quite a lot of cases, the mountable is the device name.

However this doesn't apply for btrfs subvolumes, where the mountable is a combination of both the device name and the subvolume path (see also `$g−>mountable_subvolume` to extract the subvolume path of the mountable if any).

`$subvolume` = `$g`−>mountable_subvolume ($mountable);

Returns the subvolume path of a mountable. Btrfs subvolumes mountables are a combination of both the device name and the subvolume path (see also `$g−>mountable_device` to extract the device of the mountable).

If the mountable does not represent a btrfs subvolume, then this function fails and the `errno` is set to `EINVAL`.

`%mps` = `$g`−>mountpoints ();
: This call is similar to `$g->mounts`. That call returns a list of devices. This one returns a hash table (map) of device name to directory where the device is mounted.

`@devices` = `$g`−>mounts ();
: This returns the list of currently mounted filesystems. It returns the list of devices (eg. */dev/sda1*, */dev/VG/LV*).

Some internal mounts are not shown.

See also: `$g->mountpoints`

`$g`−>mv ($src, `$dest`);
: This moves a file from `src` to `dest` where `dest` is either a destination filename or destination directory.

See also: `$g->rename`.

`$nrdisks` = `$g`−>nr_devices ();
: This returns the number of whole block devices that were added. This is the same as the number of devices that would be returned if you called `$g->list_devices`.

To find out the maximum number of devices that could be added, call `$g->max_disks`.

`$status` = `$g`−>ntfs_3g_probe ($rw, `$device`);
: This command runs the **ntfs−3g.probe** (8) command which probes an NTFS `device` for mountability. (Not all NTFS volumes can be mounted read-write, and some cannot be mounted at all).

`rw` is a boolean flag. Set it to true if you want to test if the volume can be mounted read-write. Set it to false if you want to test if the volume can be mounted read-only.

The return value is an integer which `0` if the operation would succeed, or some non-zero value documented in the **ntfs−3g.probe** (8) manual page.

This function depends on the feature `ntfs3g`. See also`$g->feature-available`.

`$g`−>ntfscat_i ($device, `$inode`, `$filename`);
: Download a file given its inode from a NTFS filesystem and save it as *filename* on the local machine.

This allows to download some otherwise inaccessible files such as the ones within the `$Extend` folder.

The filesystem from which to extract the file must be unmounted, otherwise the call will fail.

`$g`−>ntfsclone_in ($backupfile, `$device`);
: Restore the `backupfile` (from a previous call to `$g->ntfsclone_out`) to `device`, overwriting any existing contents of this device.

This function depends on the feature `ntfs3g`. See also`$g->feature-available`.

`$g`−>ntfsclone_out ($device, `$backupfile` [, metadataonly => `$metadataonly`] [, rescue => `$rescue`] [, ignorefscheck => `$ignorefscheck`] [, preservetimestamps => `$preservetimestamps`] [, force => `$force`]);
: Stream the NTFS filesystem `device` to the local file `backupfile`. The format used for the backup file is a special format used by the **ntfsclone** (8) tool.

If the optional `metadataonly` flag is true, then *only* the metadata is saved, losing all the user data (this is useful for diagnosing some filesystem problems).

The optional `rescue`, `ignorefscheck`, `preservetimestamps` and `force` flags have precise meanings detailed in the **ntfsclone** (8) man page.

Use `$g->ntfsclone_in` to restore the file back to a libguestfs device.

This function depends on the feature `ntfs3g`. See also`$g->feature-available`.

`$g`->ntfsfix ($device [, clearbadsectors => `$clearbadsectors`]);
: This command repairs some fundamental NTFS inconsistencies, resets the NTFS journal file, and schedules an NTFS consistency check for the first boot into Windows.

  This is *not* an equivalent of Windows `chkdsk`. It does*not* scan the filesystem for inconsistencies.

  The optional `clearbadsectors` flag clears the list of bad sectors. This is useful after cloning a disk with bad sectors to a new disk.

  This function depends on the feature `ntfs3g`. See also`$g->feature-available`.

`$g`->ntfsresize ($device [, size => `$size`] [, force => `$force`]);
: This command resizes an NTFS filesystem, expanding or shrinking it to the size of the underlying device.

  The optional parameters are:

  `size`
  : The new size (in bytes) of the filesystem. If omitted, the filesystem is resized to fit the container (eg. partition).

  `force`
  : If this option is true, then force the resize of the filesystem even if the filesystem is marked as requiring a consistency check.

    After the resize operation, the filesystem is always marked as requiring a consistency check (for safety). You have to boot into Windows to perform this check and clear this condition. If you *don't* set the `force` option then it is not possible to call `$g->ntfsresize` multiple times on a single filesystem without booting into Windows between each resize.

  See also **ntfsresize** (8).

  This function depends on the feature `ntfsprogs`. See also`$g->feature-available`.

`$g`->ntfsresize_opts ($device [, size => `$size`] [, force => `$force`]);
: This is an alias of ''ntfsresize''.

`$g`->ntfsresize_size ($device, `$size`);
: This command is the same as `$g->ntfsresize` except that it allows you to specify the new size (in bytes) explicitly.

  This function depends on the feature `ntfsprogs`. See also`$g->feature-available`.

  This function is deprecated. In new code, use the ''ntfsresize'' call instead.

  Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`->parse_environment ();
: Parse the programXs environment and set flags in the handle accordingly. For example if `LIBGUESTFS_DEBUG=1` then the XverboseX flag is set in the handle.

  Most programs do not need to call this. It is done implicitly when you call`$g->create`.

  See ''ENVIRONMENT VARIABLES'' in **guestfs** (3) for a list of environment variables that can affect libguestfs handles. See also ''guestfs_create_flags'' in **guestfs** (3), and `$g->parse_environment_list`.

`$g`->parse_environment_list (\@environment);
: Parse the list of strings in the argument `environment` and set flags in the handle accordingly. For example if `LIBGUESTFS_DEBUG=1` is a string in the list, then the XverboseX flag is set in the handle.

  This is the same as `$g->parse_environment` except that it parses an explicit list of strings instead of the program's environment.

$g−>part_add ($device, $prlogex, $startsect, $endsect);
: This command adds a partition to `device`. If there is no partition table on the device, call `$g−>part_init` first.

  The `prlogex` parameter is the type of partition. Normally you should pass `p` or `primary` here, but MBR partition tables also support `l` (or `logical`) and `e` (or `extended`) partition types.

  `startsect` and `endsect` are the start and end of the partition in *sectors*. `endsect` may be negative, which means it counts backwards from the end of the disk (`−1` is the last sector).

  Creating a partition which covers the whole disk is not so easy. Use`$g−>part_disk` to do that.

$g−>part_del ($device, $partnum);
: This command deletes the partition numbered `partnum` on `device`.

  Note that in the case of MBR partitioning, deleting an extended partition also deletes any logical partitions it contains.

$g−>part_disk ($device, $parttype);
: This command is simply a combination of `$g−>part_init` followed by `$g−>part_add` to create a single primary partition covering the whole disk.

  `parttype` is the partition table type, usually `mbr` or `gpt`, but other possible values are described in `$g−>part_init`.

$g−>part_expand_gpt ($device);
: Move backup GPT data structures to the end of the disk. This is useful in case of in-place image expand since disk space after backup GPT header is not usable. This is equivalent to `sgdisk −e`.

  See also **sgdisk** (8).

  This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$bootable = $g−>part_get_bootable ($device, $partnum);
: This command returns true if the partition `partnum` on `device` has the bootable flag set.

  See also `$g−>part_set_bootable`.

$guid = $g−>part_get_disk_guid ($device);
: Return the disk identifier (GUID) of a GPT-partitioned `device`. Behaviour is undefined for other partition types.

  This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$attributes = $g−>part_get_gpt_attributes ($device, $partnum);
: Return the attribute flags of numbered GPT partition `partnum`. An error is returned for MBR partitions.

  This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$guid = $g−>part_get_gpt_guid ($device, $partnum);
: Return the GUID of numbered GPT partition `partnum`.

  This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$guid = $g−>part_get_gpt_type ($device, $partnum);
: Return the type GUID of numbered GPT partition `partnum`. For MBR partitions, return an appropriate GUID corresponding to the MBR type. Behaviour is undefined for other partition types.

  This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$idbyte = $g−>part_get_mbr_id ($device, $partnum);
: Returns the MBR type byte (also known as the ID byte) from the numbered partition `partnum`.

  Note that only MBR (old DOS-style) partitions have type bytes. You will get undefined results for other partition table types (see `$g−>part_get_parttype`).

`$partitiontype` = `$g`−>part_get_mbr_part_type ($device, `$partnum`);
: This returns the partition type of an MBR partition numbered `partnum` on device `device`.

  It returns `primary`, `logical`, or `extended`.

`$name` = `$g`−>part_get_name ($device, `$partnum`);
: This gets the partition name on partition numbered `partnum` on device `device`. Note that partitions are numbered from 1.

  The partition name can only be read on certain types of partition table. This works on `gpt` but not on `mbr` partitions.

`$parttype` = `$g`−>part_get_parttype ($device);
: This command examines the partition table on `device` and returns the partition table type (format) being used.

  Common return values include: `msdos` (a DOS/Windows style MBR partition table), `gpt` (a GPT/EFI−style partition table). Other values are possible, although unusual. See `$g−>part_init` for a full list.

`$g`−>part_init ($device, `$parttype`);
: This creates an empty partition table on `device` of one of the partition types listed below. Usually `parttype` should be either `msdos` or `gpt` (for large disks).

  Initially there are no partitions. Following this, you should call `$g−>part_add` for each partition required.

  Possible values for `parttype` are:

  `efi`
  `gpt`
  : Intel EFI / GPT partition table.

    This is recommended for >= 2 TB partitions that will be accessed from Linux and Intel-based Mac OS X. It also has limited backwards compatibility with the `mbr` format.

  `mbr`
  `msdos`
  : The standard PC ''Master Boot Record'' (MBR) format used by MS-DOS and Windows. This partition type will **only** work for device sizes up to 2 TB. For large disks we recommend using `gpt`.

  Other partition table types that may work but are not supported include:

  `aix`
  : AIX disk labels.

  `amiga`
  `rdb`
  : Amiga ''Rigid Disk Block'' format.

  `bsd`
  : BSD disk labels.

  `dasd`
  : DASD, used on IBM mainframes.

  `dvh`
  : MIPS/SGI volumes.

  `mac`
  : Old Mac partition format. Modern Macs use `gpt`.

`pc98`
NEC PC−98 format, common in Japan apparently.

`sun`
Sun disk labels.

`@partitions` = `$g`−>part_list ($device);
This command parses the partition table on `device` and returns the list of partitions found.

The fields in the returned structure are:

`part_num`
Partition number, counting from 1.

`part_start`
Start of the partition *in bytes*. To get sectors you have to divide by the deviceXs sector size, see `$g->blockdev_getss`.

`part_end`
End of the partition in bytes.

`part_size`
Size of the partition in bytes.

`$g`−>part_resize ($device, `$partnum`, `$endsect`);
This command resizes the partition numbered `partnum` on `device` by moving the end position.

Note that this does not modify any filesystem present in the partition. If you wish to do this, you will need to use filesystem resizing commands like `$g−>resize2fs`.

When growing a partition you will want to grow the filesystem afterwards, but when shrinking, you need to shrink the filesystem before the partition.

`$g`−>part_set_bootable ($device, `$partnum`, `$bootable`);
This sets the bootable flag on partition numbered `partnum` on device `device`. Note that partitions are numbered from 1.

The bootable flag is used by some operating systems (notably Windows) to determine which partition to boot from. It is by no means universally recognized.

`$g`−>part_set_disk_guid ($device, `$guid`);
Set the disk identifier (GUID) of a GPT-partitioned `device` to `guid`. Return an error if the partition table of `device` isn't GPT, or if `guid` is not a valid GUID.

This function depends on the feature `gdisk`. See also`$g−>feature−available`.

`$g`−>part_set_disk_guid_random ($device);
Set the disk identifier (GUID) of a GPT-partitioned `device` to a randomly generated value. Return an error if the partition table of `device` isn't GPT.

This function depends on the feature `gdisk`. See also`$g−>feature−available`.

`$g`−>part_set_gpt_attributes ($device, `$partnum`, `$attributes`);
Set the attribute flags of numbered GPT partition `partnum` to `attributes`. Return an error if the partition table of `device` isn't GPT.

See <https://en.wikipedia.org/wiki/GUID_Partition_Table#Partition_entries> for a useful list of partition attributes.

This function depends on the feature `gdisk`. See also`$g−>feature−available`.

`$g`−>part_set_gpt_guid ($device, `$partnum`, `$guid`);
Set the GUID of numbered GPT partition `partnum` to `guid`. Return an error if the partition table of `device` isn't GPT, or if `guid` is not a valid GUID.

This function depends on the feature `gdisk`. See also`$g−>feature−available`.

$g−>part_set_gpt_type ($device, $partnum, $guid);

Set the type GUID of numbered GPT partition partnum to guid. Return an error if the partition table of device isn't GPT, or if guid is not a valid GUID.

See <https://en.wikipedia.org/wiki/GUID_Partition_Table#Partition_type_GUIDs> for a useful list of type GUIDs.

This function depends on the feature gdisk. See also$g−>feature−available.

$g−>part_set_mbr_id ($device, $partnum, $idbyte);

Sets the MBR type byte (also known as the ID byte) of the numbered partition partnum to idbyte. Note that the type bytes quoted in most documentation are in fact hexadecimal numbers, but usually documented without any leading ''0x'' which might be confusing.

Note that only MBR (old DOS-style) partitions have type bytes. You will get undefined results for other partition table types (see $g−>part_get_parttype).

$g−>part_set_name ($device, $partnum, $name);

This sets the partition name on partition numbered partnum on device device. Note that partitions are numbered from 1.

The partition name can only be set on certain types of partition table. This works on gpt but not on mbr partitions.

$device = $g−>part_to_dev ($partition);

This function takes a partition name (eg. ''/dev/sdb1'') and removes the partition number, returning the device name (eg. ''/dev/sdb'').

The named partition must exist, for example as a string returned from $g−>list_partitions.

See also $g−>part_to_partnum, $g−>device_index.

$partnum = $g−>part_to_partnum ($partition);

This function takes a partition name (eg. ''/dev/sdb1'') and returns the partition number (eg. 1).

The named partition must exist, for example as a string returned from $g−>list_partitions.

See also $g−>part_to_dev.

$g−>ping_daemon ();

This is a test probe into the guestfs daemon running inside the libguestfs appliance. Calling this function checks that the daemon responds to the ping message, without affecting the daemon or attached block device(s) in any other way.

$content = $g−>pread ($path, $count, $offset);

This command lets you read part of a file. It reads count bytes of the file, starting at offset, from file path.

This may read fewer bytes than requested. For further details see the **pread** (2) system call.

See also $g−>pwrite, $g−>pread_device.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

$content = $g−>pread_device ($device, $count, $offset);

This command lets you read part of a block device. It readscount bytes of device, starting at offset.

This may read fewer bytes than requested. For further details see the **pread** (2) system call.

See also $g−>pread.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

`$g`−>pvchange_uuid ($device);
: Generate a new random UUID for the physical volume `device`.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>pvchange_uuid_all ();
: Generate new random UUIDs for all physical volumes.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>pvcreate ($device);
: This creates an LVM physical volume on the named `device`, where `device` should usually be a partition name such as */dev/sda1*.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>pvremove ($device);
: This wipes a physical volume `device` so that LVM will no longer recognise it.

  The implementation uses the **pvremove** (8) command which refuses to wipe physical volumes that contain any volume groups, so you have to remove those first.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>pvresize ($device);
: This resizes (expands or shrinks) an existing LVM physical volume to match the new size of the underlying device.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$g`−>pvresize_size ($device, `$size`);
: This command is the same as `$g->pvresize` except that it allows you to specify the new size (in bytes) explicitly.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`@physvols` = `$g`−>pvs ();
: List all the physical volumes detected. This is the equivalent of the **pvs** (8) command.

  This returns a list of just the device names that contain PVs (eg. */dev/sda2*).

  See also `$g->pvs_full`.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`@physvols` = `$g`−>pvs_full ();
: List all the physical volumes detected. This is the equivalent of the **pvs** (8) command. The ‘‘full’’ version includes all fields.

  This function depends on the feature `lvm2`. See also`$g->feature−available`.

`$uuid` = `$g`−>pvuuid ($device);
: This command returns the UUID of the LVM PV `device`.

`$nbytes` = `$g`−>pwrite ($path, `$content`, `$offset`);
: This command writes to part of a file. It writes the data buffer `content` to the file `path` starting at offset `offset`.

  This command implements the **pwrite** (2) system call, and like that system call it may not write the full data requested. The return value is the number of bytes that were actually written to the file. This could even be 0, although short writes are unlikely for regular files in ordinary circumstances.

  See also `$g->pread`, `$g->pwrite_device`.

  Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

`$nbytes` = `$g`−>pwrite_device ($device, `$content`, `$offset`);

This command writes to part of a device. It writes the data buffer `content` to `device` starting at offset `offset`.

This command implements the **pwrite** (2) system call, and like that system call it may not write the full data requested (although short writes to disk devices and partitions are probably impossible with standard Linux kernels).

See also `$g->pwrite`.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

`$content` = `$g`−>read_file ($path);

This calls returns the contents of the file `path` as a buffer.

Unlike `$g->cat`, this function can correctly handle files that contain embedded ASCII NUL characters.

`@lines` = `$g`−>read_lines ($path);

Return the contents of the file named `path`.

The file contents are returned as a list of lines. Trailing `LF` and `CRLF` character sequences are *not* returned.

Note that this function cannot correctly handle binary files (specifically, files containing `\0` character which is treated as end of string). For those you need to use the `$g->read_file` function and split the buffer into lines yourself.

`@entries` = `$g`−>readdir ($dir);

This returns the list of directory entries in directory `dir`.

All entries in the directory are returned, including `.` and `..`. The entries are*not* sorted, b ut returned in the same order as the underlying filesystem.

Also this call returns basic file type information about each file. The `ftyp` field will contain one of the following characters:

’b’	Block special
’c’	Char special
’d’	Directory
’f’	FIFO (named pipe)
’l’	Symbolic link
’r’	Regular file
’s’	Socket
’u’	Unknown file type
’?’	The **readdir** (3) call returned a `d_type` field with an unexpected value

This function is primarily intended for use by programs. To get a simple list of names, use `$g->ls`. To get a printable directory for human consumption, use `$g->ll`.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

`$link` = `$g`−>readlink ($path);

This command reads the target of a symbolic link.

`@links` = `$g`−>readlinklist ($path, \@names);

This call allows you to do a `readlink` operation on multiple files, where all files are in the directory `path`. `names` is the list of files from this directory.

On return you get a list of strings, with a one-to-one correspondence to the `names` list. Each string is the value of the symbolic link.

If the **readlink** (2) operation fails on any name, then the corresponding result string is the empty string `""`. However the whole operation is completed even if there were **readlink** (2) errors, and so you can call this function with names where you don't know if they are symbolic links already (albeit slightly less efficient).

This call is intended for programs that want to efficiently list a directory contents without making many round-trips.

`$rpath` = `$g`−>realpath ($path);

Return the canonicalized absolute pathname of `path`. The returned path has no `.`, `..` or symbolic link path elements.

`$g`−>remount ($mountpoint [, rw => `$rw`]);

This call allows you to change the `rw` (readonly/read−write) flag on an already mounted filesystem at `mountpoint`, converting a readonly filesystem to be read-write, or vice-versa.

Note that at the moment you must supply the "optional" `rw` parameter. In future we may allow other flags to be adjusted.

`$g`−>remove_drive ($label);

This function is conceptually the opposite of `$g->add_drive_opts`. It removes the drive that was previously added with label `label`.

Note that in order to remove drives, you have to add them with labels (see the optional `label` argument to `$g->add_drive_opts`). If you didn't use a label, then they cannot be removed.

You can call this function before or after launching the handle. If called after launch, if the backend supports it, we try to hot unplug the drive: see "HOTPLUGGING" in **guestfs** (3). The disk**must not** be in use (eg. mounted) when you do this. We try to detect if the disk is in use and stop you from doing this.

`$g`−>removexattr ($xattr, `$path`);

This call removes the extended attribute named `xattr` of the file `path`.

See also: `$g->lremovexattr`, **attr** (5).

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`$g`−>rename ($oldpath, `$newpath`);

Rename a file to a new place on the same filesystem. This is the same as the Linux **rename** (2) system call. In most cases you are better to use`$g->mv` instead.

`$g`−>resize2fs ($device);

This resizes an ext2, ext3 or ext4 filesystem to match the size of the underlying device.

See also "RESIZE2FS ERRORS" in **guestfs** (3).

`$g`−>resize2fs_M ($device);

This command is the same as `$g->resize2fs`, but the filesystem is resized to its minimum size. This works like the *−M* option to the **resize2fs** (8) command.

To get the resulting size of the filesystem you should call `$g->tune2fs_l` and read the `Block size` and `Block count` values. These two numbers, multiplied together, give the resulting size of the minimal filesystem in bytes.

See also "RESIZE2FS ERRORS" in **guestfs** (3).

`$g`−>resize2fs_size ($device, `$size`);

This command is the same as `$g->resize2fs` except that it allows you to specify the new size (in bytes) explicitly.

See also "RESIZE2FS ERRORS" in **guestfs** (3).

$g−>rm ($path);
: Remove the single file `path`.

$g−>rm_f ($path);
: Remove the file `path`.

  If the file doesn't exist, that error is ignored. (Other errors, eg. I/O errors or bad paths, are not ignored)

  This call cannot remove directories. Use `$g−>rmdir` to remove an empty directory, or `$g−>rm_rf` to remove directories recursively.

$g−>rm_rf ($path);
: Remove the file or directory `path`, recursively removing the contents if its a directory. This is like the `rm −rf`shell command.

$g−>rmdir ($path);
: Remove the single directory `path`.

$g−>rmmountpoint ($exemptpath);
: This call removes a mountpoint that was previously created with `$g−>mkmountpoint`. See `$g−>mkmountpoint` for full details.

$g−>rsync ($src, `$dest` [, archive => `$archive`] [, deletedest => `$deletedest`]);
: This call may be used to copy or synchronize two directories under the same libguestfs handle. This uses the **rsync** (1) program which uses a fast algorithm that avoids copying files unnecessarily.

  `src` and `dest` are the source and destination directories. Files are copied from `src` to `dest`.

  The optional arguments are:

  `archive`
  : Turns on archive mode. This is the same as passing the −−*archive* flag to `rsync`.

  `deletedest`
  : Delete files at the destination that do not exist at the source.

  This function depends on the feature `rsync`. See also`$g−>feature−available`.

$g−>rsync_in ($remote, `$dest` [, archive => `$archive`] [, deletedest => `$deletedest`]);
: This call may be used to copy or synchronize the filesystem on the host or on a remote computer with the filesystem within libguestfs. This uses the **rsync** (1) program which uses a fast algorithm that avoids copying files unnecessarily.

  This call only works if the network is enabled. See `$g−>set_network` or the −−*network* option to various tools like **guestfish** (1).

  Files are copied from the remote server and directory specified by `remote` to the destination directory `dest`.

  The format of the remote server string is defined by **rsync** (1). Note that there is no way to supply a password or passphrase so the target must be set up not to require one.

  The optional arguments are the same as those of `$g−>rsync`.

  This function depends on the feature `rsync`. See also`$g−>feature−available`.

$g−>rsync_out ($src, `$remote` [, archive => `$archive`] [, deletedest => `$deletedest`]);
: This call may be used to copy or synchronize the filesystem within libguestfs with a filesystem on the host or on a remote computer. This uses the**rsync** (1) program which uses a f ast algorithm that avoids copying files unnecessarily.

  This call only works if the network is enabled. See `$g−>set_network` or the −−*network* option to various tools like **guestfish** (1).

  Files are copied from the source directory `src` to the remote server and directory specified by `remote`.

The format of the remote server string is defined by **rsync** (1). Note that there is no way to supply a password or passphrase so the target must be set up not to require one.

The optional arguments are the same as those of `$g->rsync`.

Globbing does not happen on the `src` parameter. In programs which use the API directly you have to expand wildcards yourself (see `$g->glob_expand`). In guestfish you can use the `glob` command (see "glob" in **guestfish** (1)), for example:

```
 ><fs> glob rsync-out /* rsync://remote/
```

This function depends on the feature `rsync`. See also `$g->feature-available`.

`$g`->scrub_device ($device);
: This command writes patterns over `device` to make data retrieval more difficult.

  It is an interface to the **scrub** (1) program. See that manual page for more details.

  This function depends on the feature `scrub`. See also `$g->feature-available`.

`$g`->scrub_file ($file);
: This command writes patterns over a file to make data retrieval more difficult.

  The file is *removed* after scrubbing.

  It is an interface to the **scrub** (1) program. See that manual page for more details.

  This function depends on the feature `scrub`. See also `$g->feature-available`.

`$g`->scrub_freespace ($dir);
: This command creates the directory `dir` and then fills it with files until the filesystem is full, and scrubs the files as for `$g->scrub_file`, and deletes them. The intention is to scrub any free space on the partition containing `dir`.

  It is an interface to the **scrub** (1) program. See that manual page for more details.

  This function depends on the feature `scrub`. See also `$g->feature-available`.

`$g`->selinux_relabel ($specfile, `$path` [, force => `$force`]);
: SELinux relabel parts of the filesystem.

  The `specfile` parameter controls the policy spec file used. You have to parse `/etc/selinux/config` to find the correct SELinux policy and then pass the spec file, usually: `/etc/selinux/` + *selinuxtype* + `/contexts/files/file_contexts`.

  The required `path` parameter is the top level directory where relabelling starts. Normally you should pass `path` as `/` to relabel the whole guest filesystem.

  The optional `force` boolean controls whether the context is reset for customizable files, and also whether the user, role and range parts of the file context is changed.

  This function depends on the feature `selinuxrelabel`. See also `$g->feature-available`.

`$g`->set_append ($append);
: This function is used to add additional options to the libguestfs appliance kernel command line.

  The default is `NULL` unless overridden by setting `LIBGUESTFS_APPEND` environment variable.

  Setting `append` to `NULL` means *no* additional options are passed (libguestfs always adds a few of its own).

`$g`->set_attach_method ($backend);
: Set the method that libguestfs uses to connect to the backend guestfsd daemon.

  See "BACKEND" in **guestfs** (3).

  This function is deprecated. In new code, use the "set_backend" call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>set_autosync ($autosync);

If `autosync` is true, this enables autosync. Libguestfs will make a best effort attempt to make filesystems consistent and synchronized when the handle is closed (also if the program exits without closing handles).

This is enabled by default (since libguestfs 1.5.24, previously it was disabled by default).

`$g`−>set_backend ($backend);

Set the method that libguestfs uses to connect to the backend guestfsd daemon.

This handle property was previously called the ‘‘attach method’’.

See ‘‘BACKEND’’ in **guestfs** (3).

`$g`−>set_backend_setting ($name, `$val`);

Append `"name=value"` to the backend settings string list. However if a string already exists matching `"name"` or beginning with `"name="`, then that setting is replaced.

See ‘‘BACKEND’’ in **guestfs** (3), ‘‘BACKEND SETTINGS’’ in **guestfs** (3).

`$g`−>set_backend_settings (\@settings);

Set a list of zero or more settings which are passed through to the current backend. Each setting is a string which is interpreted in a backend-specific way, or ignored if not understood by the backend.

The default value is an empty list, unless the environment variable `LIBGUESTFS_BACKEND_SETTINGS` was set when the handle was created. This environment variable contains a colon-separated list of settings.

This call replaces all backend settings. If you want to replace a single backend setting, see `$g−>set_backend_setting`. If you want to clear a single backend setting, see `$g−>clear_backend_setting`.

See ‘‘BACKEND’’ in **guestfs** (3), ‘‘BACKEND SETTINGS’’ in **guestfs** (3).

`$g`−>set_cachedir ($cachedir);

Set the directory used by the handle to store the appliance cache, when using a supermin appliance. The appliance is cached and shared between all handles which have the same effective user ID.

The environment variables `LIBGUESTFS_CACHEDIR` and `TMPDIR` control the default value: If `LIBGUESTFS_CACHEDIR` is set, then that is the default. Else if`TMPDIR` is set, then that is the default. Else*/var/tmp* is the def ault.

`$g`−>set_direct ($direct);

If the direct appliance mode flag is enabled, then stdin and stdout are passed directly through to the appliance once it is launched.

One consequence of this is that log messages aren’t caught by the library and handled by `$g−>set_log_message_callback`, but go straight to stdout.

You probably don’t want to use this unless you know what you are doing.

The default is disabled.

This function is deprecated. In new code, use the ‘‘internal_get_console_socket’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>set_e2attrs ($file, `$attrs` [, clear => `$clear`]);

This sets or clears the file attributes `attrs` associated with the inode *file*.

`attrs` is a string of characters representing file attributes. See`$g−>get_e2attrs` for a list of possible attributes. Not all attributes can be changed.

If optional boolean `clear` is not present or false, then the `attrs` listed are set in the inode.

If `clear` is true, then the `attrs` listed are cleared in the inode.

In both cases, other attributes not present in the `attrs` string are left unchanged.

These attributes are only present when the file is located on an ext2/3/4 filesystem. Using this call on other filesystem types will result in an error.

`$g`−>set_e2generation ($file, `$generation`);

This sets the ext2 file generation of a file.

See `$g->get_e2generation`.

`$g`−>set_e2label ($device, `$label`);

This sets the ext2/3/4 filesystem label of the filesystem on `device` to `label`. Filesystem labels are limited to 16 characters.

You can use either `$g->tune2fs_l` or `$g->get_e2label` to return the existing label on a filesystem.

This function is deprecated. In new code, use the ‘‘set_label’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>set_e2uuid ($device, `$uuid`);

This sets the ext2/3/4 filesystem UUID of the filesystem on `device` to `uuid`. The format of the UUID and alternatives such as `clear`, `random` and `time` are described in the **tune2fs** (8) manpage.

You can use `$g->vfs_uuid` to return the existing UUID of a filesystem.

This function is deprecated. In new code, use the ‘‘set_uuid’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>set_hv ($hv);

Set the hypervisor binary that we will use. The hypervisor depends on the backend, but is usually the location of the qemu/KVM hypervisor. For the uml backend, it is the location of the `linux` or `vmlinux` binary.

The default is chosen when the library was compiled by the configure script.

You can also override this by setting the `LIBGUESTFS_HV` environment variable.

Note that you should call this function as early as possible after creating the handle. This is because some pre-launch operations depend on testing qemu features (by running `qemu -help`). If the qemu binary changes, we don't retest features, and so you might see inconsistent results. Using the environment variable `LIBGUESTFS_HV` is safest of all since that picks the qemu binary at the same time as the handle is created.

`$g`−>set_identifier ($identifier);

This is an informative string which the caller may optionally set in the handle. It is printed in various places, allowing the current handle to be identified in debugging output.

One important place is when tracing is enabled. If the identifier string is not an empty string, then trace messages change from this:

```
 libguestfs: trace: get_tmpdir
 libguestfs: trace: get_tmpdir = "/tmp"
```

to this:

```
 libguestfs: trace: ID: get_tmpdir
 libguestfs: trace: ID: get_tmpdir = "/tmp"
```

where `ID` is the identifier string set by this call.

The identifier must only contain alphanumeric ASCII characters, underscore and minus sign. The default is the empty string.

See also `$g->set_program`, `$g->set_trace`, `$g->get_identifier`.

`$g`−>set_label ($mountable, `$label`);

Set the filesystem label on `mountable` to `label`.

Only some filesystem types support labels, and libguestfs supports setting labels on only a subset of these.

ext2, ext3, ext4
: Labels are limited to 16 bytes.

NTFS
: Labels are limited to 128 unicode characters.

XFS
: The label is limited to 12 bytes. The filesystem must not be mounted when trying to set the label.

btrfs
: The label is limited to 255 bytes and some characters are not allowed. Setting the label on a btrfs subvolume will set the label on its parent filesystem. The filesystem must not be mounted when trying to set the label.

fat
: The label is limited to 11 bytes.

swap
: The label is limited to 16 bytes.

If there is no support for changing the label for the type of the specified filesystem, set_label will fail and set errno as ENOTSUP.

To read the label on a filesystem, call `$g->vfs_label`.

`$g`−>set_libvirt_requested_credential ($index, `$cred`);

After requesting the `index`'th credential from the user, call this function to pass the answer back to libvirt.

See ''LIBVIRT AUTHENTICATION'' in **guestfs** (3) for documentation and example code.

`$g`−>set_libvirt_supported_credentials (\@creds);

Call this function before setting an event handler for `GUESTFS_EVENT_LIBVIRT_AUTH`, to supply the list of credential types that the program knows how to process.

The `creds` list must be a non-empty list of strings. Possible strings are:

```
username
authname
language
cnonce
passphrase
echoprompt
noechoprompt
realm
external
```

See libvirt documentation for the meaning of these credential types.

See ''LIBVIRT AUTHENTICATION'' in **guestfs** (3) for documentation and example code.

$g−>set_memsize ($memsize);

This sets the memory size in megabytes allocated to the hypervisor. This only has any effect if called before `$g−>launch`.

You can also change this by setting the environment variable `LIBGUESTFS_MEMSIZE` before the handle is created.

For more information on the architecture of libguestfs, see **guestfs** (3).

$g−>set_network ($network);

If `network` is true, then the network is enabled in the libguestfs appliance. The default is false.

This affects whether commands are able to access the network (see ''RUNNING COMMANDS'' in **guestfs** (3)).

You must call this before calling `$g−>launch`, otherwise it has no effect.

$g−>set_path ($searchpath);

Set the path that libguestfs searches for kernel and initrd.img.

The default is `$libdir/guestfs` unless overridden by setting `LIBGUESTFS_PATH` environment variable.

Setting `path` to `NULL` restores the default path.

$g−>set_pgroup ($pgroup);

If `pgroup` is true, child processes are placed into their own process group.

The practical upshot of this is that signals like `SIGINT` (from users pressing `^C`) won't be received by the child process.

The default for this flag is false, because usually you want `^C` to kill the subprocess. Guestfish sets this flag to true when used interactively, so that `^C` can cancel long-running commands gracefully (see `$g−>user_cancel`).

$g−>set_program ($program);

Set the program name. This is an informative string which the main program may optionally set in the handle.

When the handle is created, the program name in the handle is set to the basename from `argv[0]`. The program name can never be `NULL`.

$g−>set_qemu ($hv);

Set the hypervisor binary (usually qemu) that we will use.

The default is chosen when the library was compiled by the configure script.

You can also override this by setting the `LIBGUESTFS_HV` environment variable.

Setting `hv` to `NULL` restores the default qemu binary.

Note that you should call this function as early as possible after creating the handle. This is because some pre-launch operations depend on testing qemu features (by running `qemu −help`). If the qemu binary changes, we don't retest features, and so you might see inconsistent results. Using the environment variable `LIBGUESTFS_HV` is safest of all since that picks the qemu binary at the same time as the handle is created.

This function is deprecated. In new code, use the ''set_hv'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>set_recovery_proc ($recoveryproc);

If this is called with the parameter `false` then `$g−>launch` does not create a recovery process. The purpose of the recovery process is to stop runaway hypervisor processes in the case where the main program aborts abruptly.

This only has any effect if called before `$g->launch`, and the default is true.

About the only time when you would want to disable this is if the main process will fork itself into the background (‘‘daemonize’’ itself). In this case the recovery process thinks that the main program has disappeared and so kills the hypervisor, which is not very helpful.

`$g`−>set_selinux ($selinux);

This sets the selinux flag that is passed to the appliance at boot time. The default is `selinux=0` (disabled).

Note that if SELinux is enabled, it is always in Permissive mode (`enforcing=0`).

For more information on the architecture of libguestfs, see **guestfs** (3).

This function is deprecated. In new code, use the ‘‘selinux_relabel’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>set_smp ($smp);

Change the number of virtual CPUs assigned to the appliance. The default is `1`. Increasing this may improve performance, though often it has no effect.

This function must be called before `$g->launch`.

`$g`−>set_tmpdir ($tmpdir);

Set the directory used by the handle to store temporary files.

The environment variables `LIBGUESTFS_TMPDIR` and `TMPDIR` control the default value: If `LIBGUESTFS_TMPDIR` is set, then that is the default. Else if`TMPDIR` is set, then that is the def ault. Else */tmp* is the default.

`$g`−>set_trace ($trace);

If the command trace flag is set to 1, then libguestfs calls, parameters and return values are traced.

If you want to trace C API calls into libguestfs (and other libraries) then possibly a better way is to use the external **ltrace** (1) command.

Command traces are disabled unless the environment variable `LIBGUESTFS_TRACE` is defined and set to `1`.

Trace messages are normally sent to `stderr`, unless you register a callback to send them somewhere else (see `$g->set_event_callback`).

`$g`−>set_uuid ($device, `$uuid`);

Set the filesystem UUID on `device` to `uuid`. If this fails and the errno is ENOTSUP, means that there is no support for changing the UUID for the type of the specified filesystem.

Only some filesystem types support setting UUIDs.

To read the UUID on a filesystem, call `$g->vfs_uuid`.

`$g`−>set_uuid_random ($device);

Set the filesystem UUID on `device` to a random UUID. If this fails and the errno is ENOTSUP, means that there is no support for changing the UUID for the type of the specified filesystem.

Only some filesystem types support setting UUIDs.

To read the UUID on a filesystem, call `$g->vfs_uuid`.

`$g`−>set_verbose ($verbose);

If `verbose` is true, this turns on verbose messages.

Verbose messages are disabled unless the environment variable `LIBGUESTFS_DEBUG` is defined and set to `1`.

Verbose messages are normally sent to `stderr`, unless you register a callback to send them somewhere else (see `$g->set_event_callback`).

`$g`−>setcon ($context);

This sets the SELinux security context of the daemon to the string `context`.

See the documentation about SELINUX in **guestfs** (3).

This function depends on the feature `selinux`. See also`$g->feature-available`.

This function is deprecated. In new code, use the ''selinux_relabel'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>setxattr ($xattr, `$val`, `$vallen`, `$path`);

This call sets the extended attribute named `xattr` of the file `path` to the value `val` (of length `vallen`). The value is arbitrary 8 bit data.

See also: `$g->lsetxattr`, **attr** (5).

This function depends on the feature `linuxxattrs`. See also`$g->feature-available`.

`$g`−>sfdisk ($device, `$cyls`, `$heads`, `$sectors`, \@lines);

This is a direct interface to the **sfdisk** (8) program for creating partitions on block devices.

`device` should be a block device, for example */dev/sda*.

`cyls`, `heads` and `sectors` are the number of cylinders, heads and sectors on the device, which are passed directly to **sfdisk** (8) as the *−C*, *−H* and *−S* parameters. If you pass `0` for any of these, then the corresponding parameter is omitted. Usually for XlargeX disks, you can just pass `0` for these, but for small (floppy-sized) disks, **sfdisk** (8) (or rather, the kernel) cannot work out the right geometry and you will need to tell it.

`lines` is a list of lines that we feed to **sfdisk** (8). For more information refer to the **sfdisk** (8) manpage.

To create a single partition occupying the whole disk, you would pass `lines` as a single element list, when the single element being the string `,` (comma).

See also: `$g->sfdisk_l`, `$g->sfdisk_N`, `$g->part_init`

This function is deprecated. In new code, use the ''part_add'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>sfdiskM ($device, \@lines);

This is a simplified interface to the `$g->sfdisk` command, where partition sizes are specified in megabytes only (rounded to the nearest cylinder) and you don't need to specify the cyls, heads and sectors parameters which were rarely if ever used anyway.

See also: `$g->sfdisk`, the **sfdisk** (8) manpage and `$g->part_disk`

This function is deprecated. In new code, use the ''part_add'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>sfdisk_N ($device, `$partnum`, `$cyls`, `$heads`, `$sectors`, `$line`);

This runs **sfdisk** (8) option to modify just the single partition `n` (note: `n` counts from 1).

For other parameters, see `$g->sfdisk`. You should usually pass `0` for the cyls/heads/sectors parameters.

See also: `$g->part_add`

This function is deprecated. In new code, use the ''part_add'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$partitions` = `$g`−>sfdisk_disk_geometry ($device);

This displays the disk geometry of `device` read from the partition table. Especially in the case where the underlying block device has been resized, this can be different from the kernelXs idea of the geometry (see `$g->sfdisk_kernel_geometry`).

The result is in human-readable format, and not designed to be parsed.

`$partitions` = `$g`−>sfdisk_kernel_geometry ($device);

This displays the kernelXs idea of the geometry of `device`.

The result is in human-readable format, and not designed to be parsed.

`$partitions` = `$g`−>sfdisk_l ($device);

This displays the partition table on `device`, in the human-readable output of the **sfdisk** (8) command. It is not intended to be parsed.

See also: `$g->part_list`

This function is deprecated. In new code, use the ‘‘part_list’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$output` = `$g`−>sh ($command);

This call runs a command from the guest filesystem via the guestXs */bin/sh*.

This is like `$g->command`, but passes the command to:

```
 /bin/sh -c "command"
```

Depending on the guestXs shell, this usually results in wildcards being expanded, shell expressions being interpolated and so on.

All the provisos about `$g->command` apply to this call.

`@lines` = `$g`−>sh_lines ($command);

This is the same as `$g->sh`, but splits the result into a list of lines.

See also: `$g->command_lines`

`$g`−>shutdown ();

This is the opposite of `$g->launch`. It performs an orderly shutdown of the backend process(es). If the autosync flag is set (which is the default) then the disk image is synchronized.

If the subprocess exits with an error then this function will return an error, which should *not* be ignored (it may indicate that the disk image could not be written out properly).

It is safe to call this multiple times. Extra calls are ignored.

This call does *not* close or free up the handle. You still need to call `$g->close` afterwards.

`$g->close` will call this if you don’t do it explicitly, but note that any errors are ignored in that case.

`$g`−>sleep ($secs);

Sleep for `secs` seconds.

`%statbuf` = `$g`−>stat ($path);

Returns file information for the given `path`.

This is the same as the **stat** (2) system call.

This function is deprecated. In new code, use the ‘‘statns’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`%statbuf` = `$g`−>statns ($path);

Returns file information for the given `path`.

This is the same as the **stat** (2) system call.

`%statbuf` = `$g`−>statvfs ($path);

Returns file system statistics for any mounted file system. `path` should be a file or directory in the mounted file system (typically it is the mount point itself, but it doesn't need to be).

This is the same as the **statvfs** (2) system call.

`@stringsout` = `$g`−>strings ($path);

This runs the **strings** (1) command on a file and returns the list of printable strings found.

The `strings` command has, in the past, had problems with parsing untrusted files. These are mitigated in the current version of libguestfs, but see "CVE−2014−8484" in **guestfs** (3).

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

`@stringsout` = `$g`−>strings_e ($encoding, `$path`);

This is like the `$g−>strings` command, but allows you to specify the encoding of strings that are looked for in the source file `path`.

Allowed encodings are:

s
: Single 7−bit−byte characters like ASCII and the ASCII-compatible parts of ISO−8859−X (this is what `$g−>strings` uses).

S
: Single 8−bit−byte characters.

b
: 16−bit big endian strings such as those encoded in UTF−16BE or UCS−2BE.

l (lower case letter L)
: 16−bit little endian such as UTF−16LE and UCS−2LE. This is useful for examining binaries in Windows guests.

B
: 32−bit big endian such as UCS−4BE.

L
: 32−bit little endian such as UCS−4LE.

The returned strings are transcoded to UTF−8.

The `strings` command has, in the past, had problems with parsing untrusted files. These are mitigated in the current version of libguestfs, but see "CVE−2014−8484" in **guestfs** (3).

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See "PROTOCOL LIMITS" in **guestfs** (3).

`$g`−>swapoff_device ($device);

This command disables the libguestfs appliance swap device or partition named `device`. See `$g−>swapon_device`.

`$g`−>swapoff_file ($file);

This command disables the libguestfs appliance swap on file.

`$g`−>swapoff_label ($label);

This command disables the libguestfs appliance swap on labeled swap partition.

`$g`−>swapoff_uuid ($uuid);

This command disables the libguestfs appliance swap partition with the given UUID.

This function depends on the feature `linuxfsuuid`. See also`$g−>feature−available`.

`$g`−>swapon_device ($device);

This command enables the libguestfs appliance to use the swap device or partition named `device`. The increased memory is made available for all commands, for example those run using

`$g->command` or `$g->sh`.

Note that you should not swap to existing guest swap partitions unless you know what you are doing. They may contain hibernation information, or other information that the guest doesn't want you to trash. You also risk leaking information about the host to the guest this way. Instead, attach a new host device to the guest and swap on that.

$g->swapon_file ($file);
: This command enables swap to a file. See `$g->swapon_device` for other notes.

$g->swapon_label ($label);
: This command enables swap to a labeled swap partition. See `$g->swapon_device` for other notes.

$g->swapon_uuid ($uuid);
: This command enables swap to a swap partition with the given UUID. See`$g->swapon_device` for other notes.

This function depends on the feature `linuxfsuuid`. See also`$g->feature-available`.

$g->sync ();
: This syncs the disk, so that any writes are flushed through to the underlying disk image.

You should always call this if you have modified a disk image, before closing the handle.

$g->syslinux ($device [, directory => $directory]);
: Install the SYSLINUX bootloader on `device`.

The device parameter must be either a whole disk formatted as a FAT filesystem, or a partition formatted as a FAT filesystem. In the latter case, the partition should be marked as ''active'' (`$g->part_set_bootable`) and a Master Boot Record must be installed (eg. using `$g->pwrite_device`) on the first sector of the whole disk. The SYSLINUX package comes with some suitable Master Boot Records. See the **syslinux** (1) man page for further information.

The optional arguments are:

directory
: Install SYSLINUX in the named subdirectory, instead of in the root directory of the FAT filesystem.

Additional configuration can be supplied to SYSLINUX by placing a file called *syslinux.cfg* on the FAT filesystem, either in the root directory, or under *directory* if that optional argument is being used. For further information about the contents of this file, see **syslinux** (1).

See also `$g->extlinux`.

This function depends on the feature `syslinux`. See also`$g->feature-available`.

@lines = $g->tail ($path);
: This command returns up to the last 10 lines of a file as a list of strings.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

@lines = $g->tail_n ($nrlines, $path);
: If the parameter `nrlines` is a positive number, this returns the last `nrlines` lines of the file `path`.

If the parameter `nrlines` is a negative number, this returns lines from the file `path`, starting with the `-nrlines`'th line.

If the parameter `nrlines` is zero, this returns an empty list.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

`$g`−>tar_in ($tarfile, `$directory` [, compress => `$compress`] [, xattrs => `$xattrs`] [, selinux => `$selinux`] [, acls => `$acls`]);

This command uploads and unpacks local file `tarfile` into *directory*.

The optional `compress` flag controls compression. If not given, then the input should be an uncompressed tar file. Otherwise one of the following strings may be given to select the compression type of the input file: `compress`, `gzip`, `bzip2`, `xz`, `lzop`. (Note that not all builds of libguestfs will support all of these compression types).

The other optional arguments are:

`xattrs`
If set to true, extended attributes are restored from the tar file.

`selinux`
If set to true, SELinux contexts are restored from the tar file.

`acls`
If set to true, POSIX ACLs are restored from the tar file.

`$g`−>tar_in_opts ($tarfile, `$directory` [, compress => `$compress`] [, xattrs => `$xattrs`] [, selinux => `$selinux`] [, acls => `$acls`]);

This is an alias of ''tar_in''.

`$g`−>tar_out ($directory, `$tarfile` [, compress => `$compress`] [, numericowner => `$numericowner`] [, excludes => `$excludes`] [, xattrs => `$xattrs`] [, selinux => `$selinux`] [, acls => `$acls`]);

This command packs the contents of *directory* and downloads it to local file `tarfile`.

The optional `compress` flag controls compression. If not given, then the output will be an uncompressed tar file. Otherwise one of the following strings may be given to select the compression type of the output file: `compress`, `gzip`, `bzip2`, `xz`, `lzop`. (Note that not all builds of libguestfs will support all of these compression types).

The other optional arguments are:

`excludes`
A list of wildcards. Files are excluded if they match any of the wildcards.

`numericowner`
If set to true, the output tar file will contain UID/GID numbers instead of user/group names.

`xattrs`
If set to true, extended attributes are saved in the output tar.

`selinux`
If set to true, SELinux contexts are saved in the output tar.

`acls`
If set to true, POSIX ACLs are saved in the output tar.

`$g`−>tar_out_opts ($directory, `$tarfile` [, compress => `$compress`] [, numericowner => `$numericowner`] [, excludes => `$excludes`] [, xattrs => `$xattrs`] [, selinux => `$selinux`] [, acls => `$acls`]);

This is an alias of ''tar_out''.

`$g`−>tgz_in ($tarball, `$directory`);

This command uploads and unpacks local file `tarball` (a *gzip compressed* tar file) into *directory*.

This function is deprecated. In new code, use the ''tar_in'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>tgz_out ($directory, `$tarball`);

This command packs the contents of *directory* and downloads it to local file `tarball`.

This function is deprecated. In new code, use the ''tar_out'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>touch ($path);

Touch acts like the **touch** (1) command. It can be used to update the timestamps on a file, or, if the file does not exist, to create a new zero-length file.

This command only works on regular files, and will fail on other file types such as directories, symbolic links, block special etc.

`$g`−>truncate ($path);

This command truncates `path` to a zero-length file. The file must exist already.

`$g`−>truncate_size ($path, `$size`);

This command truncates `path` to size `size` bytes. The file must exist already.

If the current file size is less than `size` then the file is extended to the required size with zero bytes. This creates a sparse file (ie. disk blocks are not allocated for the file until you write to it). To create a non-sparse file of zeroes, use `$g−>fallocate64` instead.

`$g`−>tune2fs ($device [, force => `$force`] [, maxmountcount => `$maxmountcount`] [, mountcount => `$mountcount`] [, errorbehavior => `$errorbehavior`] [, group => `$group`] [, intervalbetweenchecks => `$intervalbetweenchecks`] [, reservedblockspercentage => `$reservedblockspercentage`] [, lastmounteddirectory => `$lastmounteddirectory`] [, reservedblockscount => `$reservedblockscount`] [, user => `$user`]);

This call allows you to adjust various filesystem parameters of an ext2/ext3/ext4 filesystem called `device`.

The optional parameters are:

`force`

Force tune2fs to complete the operation even in the face of errors. This is the same as the **tune2fs** (8) `−f` option.

`maxmountcount`

Set the number of mounts after which the filesystem is checked by **e2fsck** (8). If this is`0` then the number of mounts is disregarded. This is the same as the**tune2fs** (8) `−c` option.

`mountcount`

Set the number of times the filesystem has been mounted. This is the same as the **tune2fs** (8) `−C` option.

`errorbehavior`

Change the behavior of the kernel code when errors are detected. Possible values currently are: `continue`, `remount−ro`, `panic`. In practice these options don't really make any difference, particularly for write errors.

This is the same as the **tune2fs** (8) `−e` option.

`group`

Set the group which can use reserved filesystem blocks. This is the same as the **tune2fs** (8) `−g` option except that it can only be specified as a number.

`intervalbetweenchecks`

Adjust the maximal time between two filesystem checks (in seconds). If the option is passed as `0` then time-dependent checking is disabled.

This is the same as the **tune2fs** (8) `−i` option.

`reservedblockspercentage`
Set the percentage of the filesystem which may only be allocated by privileged processes. This is the same as the **tune2fs** (8) `-m` option.

`lastmounteddirectory`
Set the last mounted directory. This is the same as the**tune2fs** (8) `-M` option.

`reservedblockscount` Set the number of reserved filesystem blocks. This is the same as the **tune2fs** (8) `-r` option.

`user`
Set the user who can use the reserved filesystem blocks. This is the same as the **tune2fs** (8) `-u` option except that it can only be specified as a number.

To get the current values of filesystem parameters, see `$g->tune2fs_l`. For precise details of how tune2fs works, see the **tune2fs** (8) man page.

`%superblock` = `$g`->tune2fs_l ($device);
This returns the contents of the ext2, ext3 or ext4 filesystem superblock on `device`.

It is the same as running `tune2fs -l device`. See**tune2fs** (8) manpage for more details. The list of fields returned isn't clearly defined, and depends on both the version of `tune2fs` that libguestfs was built against, and the filesystem itself.

`$g`->txz_in ($tarball, `$directory`);
This command uploads and unpacks local file `tarball` (an *xz compressed* tar file) into *directory*.

This function depends on the feature `xz`. See also`$g->feature-available`.

This function is deprecated. In new code, use the ‘‘tar_in’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`->txz_out ($directory, `$tarball`);
This command packs the contents of *directory* and downloads it to local file `tarball` (as an xz compressed tar archive).

This function depends on the feature `xz`. See also`$g->feature-available`.

This function is deprecated. In new code, use the ‘‘tar_out’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$oldmask` = `$g`->umask ($mask);
This function sets the mask used for creating new files and device nodes to `mask & 0777`.

Typical umask values would be `022` which creates new files with permissions like ‘‘−rw−r — r−−’’ or ‘‘−rwxr−xr−x’’, and `002` which creates new files with permissions like ‘‘−rw−rw−r−−’’ or ‘‘−rwxrwxr−x’’.

The default umask is `022`. This is important because it means that directories and device nodes will be created with `0644` or `0755` mode even if you specify `0777`.

See also `$g->get_umask`, **umask** (2), `$g->mknod`, `$g->mkdir`.

This call returns the previous umask.

`$g`->umount ($pathordevice [, force => `$force`] [, lazyunmount => `$lazyunmount`]);
This unmounts the given filesystem. The filesystem may be specified either by its mountpoint (path) or the device which contains the filesystem.

`$g`->umount_opts ($pathordevice [, force => `$force`] [, lazyunmount => `$lazyunmount`]);
This is an alias of ‘‘umount’’.

$g−>umount_all ();

This unmounts all mounted filesystems.

Some internal mounts are not unmounted by this call.

$g−>umount_local ([retry => $retry]);

If libguestfs is exporting the filesystem on a local mountpoint, then this unmounts it.

See ''MOUNT LOCAL'' in **guestfs** (3) for full documentation.

$g−>upload ($filename, $remotefilename);

Upload local file *filename* to *remotefilename* on the filesystem.

filename can also be a named pipe.

See also `$g−>download`.

$g−>upload_offset ($filename, $remotefilename, $offset);

Upload local file *filename* to *remotefilename* on the filesystem.

remotefilename is overwritten starting at the byte `offset` specified. The intention is to overwrite parts of existing files or devices, although if a non-existent file is specified then it is created with a ''hole'' before `offset`. The size of the data written is implicit in the size of the source*filename*.

Note that there is no limit on the amount of data that can be uploaded with this call, unlike with `$g−>pwrite`, and this call always writes the full amount unless an error occurs.

See also `$g−>upload`, `$g−>pwrite`.

$g−>user_cancel ();

This function cancels the current upload or download operation.

Unlike most other libguestfs calls, this function is signal safe and thread safe. You can call it from a signal handler or from another thread, without needing to do any locking.

The transfer that was in progress (if there is one) will stop shortly afterwards, and will return an error. The errno (see ''guestfs_last_errno'') is set to `EINTR`, so you can test for this to find out if the operation was cancelled or failed because of another error.

No cleanup is performed: for example, if a file was being uploaded then after cancellation there may be a partially uploaded file. It is the callerXs responsibility to clean up if necessary.

There are two common places that you might call `$g−>user_cancel`:

In an interactive text-based program, you might call it from a `SIGINT` signal handler so that pressing `ˆC` cancels the current operation. (You also need to call `$g−>set_pgroup` so that child processes don't receive the `ˆC` signal).

In a graphical program, when the main thread is displaying a progress bar with a cancel button, wire up the cancel button to call this function.

$g−>utimens ($path, $atsecs, $atnsecs, $mtsecs, $mtnsecs);

This command sets the timestamps of a file with nanosecond precision.

`atsecs`, `atnsecs` are the last access time (atime) in secs and nanoseconds from the epoch.

`mtsecs`, `mtnsecs` are the last modification time (mtime) in secs and nanoseconds from the epoch.

If the `*nsecs` field contains the special value `−1` then the corresponding timestamp is set to the current time. (The `*secs` field is ignored in this case).

If the `*nsecs` field contains the special value `−2` then the corresponding timestamp is left unchanged. (The `*secs` field is ignored in this case).

%uts = $g−>utsname ();

This returns the kernel version of the appliance, where this is available. This information is only useful for debugging. Nothing in the returned structure is defined by the API.

`%version` = `$g`−>version ();
: Return the libguestfs version number that the program is linked against.

Note that because of dynamic linking this is not necessarily the version of libguestfs that you compiled against. You can compile the program, and then at runtime dynamically link against a completely different *libguestfs.so* library.

This call was added in version `1.0.58`. In previous versions of libguestfs there was no way to get the version number. From C code you can use dynamic linker functions to find out if this symbol exists (if it doesn't, then itXs an earlier version).

The call returns a structure with four elements. The first three (`major`, `minor` and `release`) are numbers and correspond to the usual version triplet. The fourth element (`extra`) is a string and is normally empty, but may be used for distro-specific information.

To construct the original version string: `$major.$minor.$release$extra`

See also: ''LIBGUESTFS VERSION NUMBERS'' in **guestfs** (3).

Note: Don't use this call to test for availability of features. In enterprise distributions we backport features from later versions into earlier versions, making this an unreliable way to test for features. Use `$g->available` or `$g->feature_available` instead.

`$label` = `$g`−>vfs_label ($mountable);
: This returns the label of the filesystem on `mountable`.

If the filesystem is unlabeled, this returns the empty string.

To find a filesystem from the label, use `$g->findfs_label`.

`$sizeinbytes` = `$g`−>vfs_minimum_size ($mountable);
: Get the minimum size of filesystem in bytes. This is the minimum possible size for filesystem shrinking.

If getting minimum size of specified filesystem is not supported, this will fail and set errno as ENOTSUP.

See also **ntfsresize** (8), **resize2fs** (8), **btrfs** (8), **xfs_info** (8).

`$fstype` = `$g`−>vfs_type ($mountable);
: This command gets the filesystem type corresponding to the filesystem on `mountable`.

For most filesystems, the result is the name of the Linux VFS module which would be used to mount this filesystem if you mounted it without specifying the filesystem type. For example a string such as `ext3` or `ntfs`.

`$uuid` = `$g`−>vfs_uuid ($mountable);
: This returns the filesystem UUID of the filesystem on `mountable`.

If the filesystem does not have a UUID, this returns the empty string.

To find a filesystem from the UUID, use `$g->findfs_uuid`.

`$g`−>vg_activate ($activate, \@volgroups);
: This command activates or (if `activate` is false) deactivates all logical volumes in the listed volume groups `volgroups`.

This command is the same as running `vgchange -a y|n volgroups...`

Note that if `volgroups` is an empty list then **all** volume groups are activated or deactivated.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vg_activate_all ($activate);
: This command activates or (if `activate` is false) deactivates all logical volumes in all volume groups.

This command is the same as running `vgchange -a y|n`

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vgchange_uuid ($vg);
: Generate a new random UUID for the volume group `vg`.

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vgchange_uuid_all ();
: Generate new random UUIDs for all volume groups.

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vgcreate ($volgroup, \@physvols);
: This creates an LVM volume group called `volgroup` from the non-empty list of physical volumes `physvols`.

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`@uuids` = `$g`−>vglvuuids ($vgname);
: Given a VG called `vgname`, this returns the UUIDs of all the logical volumes created in this volume group.

  You can use this along with `$g->lvs` and `$g->lvuuid` calls to associate logical volumes and volume groups.

  See also `$g->vgpvuuids`.

`$metadata` = `$g`−>vgmeta ($vgname);
: `vgname` is an LVM volume group. This command examines the volume group and returns its metadata.

  Note that the metadata is an internal structure used by LVM, subject to change at any time, and is provided for information only.

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`@uuids` = `$g`−>vgpvuuids ($vgname);
: Given a VG called `vgname`, this returns the UUIDs of all the physical volumes that this volume group resides on.

  You can use this along with `$g->pvs` and `$g->pvuuid` calls to associate physical volumes and volume groups.

  See also `$g->vglvuuids`.

`$g`−>vgremove ($vgname);
: Remove an LVM volume group `vgname`, (for example `VG`).

  This also forcibly removes all logical volumes in the volume group (if any).

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vgrename ($volgroup, `$newvolgroup`);
: Rename a volume group `volgroup` with the new name `newvolgroup`.

`@volgroups` = `$g`−>vgs ();
: List all the volumes groups detected. This is the equivalent of the **vgs**(8) command.

  This returns a list of just the volume group names that were detected (eg. `VolGroup00`).

  See also `$g->vgs_full`.

  This function depends on the feature `lvm2`. See also`$g->feature-available`.

`@volgroups` = `$g`−>vgs_full ();

List all the volumes groups detected. This is the equivalent of the **vgs**(8) command. The ''full'' version includes all fields.

This function depends on the feature `lvm2`. See also`$g->feature-available`.

`$g`−>vgscan ();

This rescans all block devices and rebuilds the list of LVM physical volumes, volume groups and logical volumes.

This function is deprecated. In new code, use the ''lvm_scan'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$uuid` = `$g`−>vguuid ($vgname);

This command returns the UUID of the LVM VG named `vgname`.

`$g`−>wait_ready ();

This function is a no op.

In versions of the API < 1.0.71 you had to call this function just after calling `$g->launch` to wait for the launch to complete. However this is no longer necessary because `$g->launch` now does the waiting.

If you see any calls to this function in code then you can just remove them, unless you want to retain compatibility with older versions of the API.

This function is deprecated. There is no replacement. Consult the API documentation in **guestfs**(3) for further information.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$chars` = `$g`−>wc_c ($path);

This command counts the characters in a file, using the `wc -c` external command.

`$lines` = `$g`−>wc_l ($path);

This command counts the lines in a file, using the `wc -l` external command.

`$words` = `$g`−>wc_w ($path);

This command counts the words in a file, using the `wc -w` external command.

`$g`−>wipefs ($device);

This command erases filesystem or RAID signatures from the specified `device` to make the filesystem invisible to libblkid.

This does not erase the filesystem itself nor any other data from the `device`.

Compare with `$g->zero` which zeroes the first few blocks of a device.

This function depends on the feature `wipefs`. See also`$g->feature-available`.

`$g`−>write ($path, `$content`);

This call creates a file called `path`. The content of the file is the string`content` (which can contain any 8 bit data).

See also `$g->write_append`.

`$g`−>write_append ($path, `$content`);

This call appends `content` to the end of file `path`. If`path` does not e xist, then a new file is created.

See also `$g->write`.

$g−>write_file ($path, $content, $size);

This call creates a file called `path`. The contents of the file is the string`content` (which can contain any 8 bit data), with length `size`.

As a special case, if `size` is `0` then the length is calculated using `strlen` (so in this case the content cannot contain embedded ASCII NULs).

NB. Owing to a bug, writing content containing ASCII NUL characters does *not* work, even if the length is specified.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘write’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

$g−>xfs_admin ($device [, extunwritten => $extunwritten] [, imgfile => $imgfile] [, v2log => $v2log] [, projid32bit => $projid32bit] [, lazycounter => $lazycounter] [, label => $label] [, uuid => $uuid]);

Change the parameters of the XFS filesystem on `device`.

Devices that are mounted cannot be modified. Administrators must unmount filesystems before this call can modify parameters.

Some of the parameters of a mounted filesystem can be examined and modified using the `$g−>xfs_info` and `$g−>xfs_growfs` calls.

Beginning with XFS version 5, it is no longer possible to modify the lazy-counters setting (ie. `lazycounter` parameter has no effect).

This function depends on the feature `xfs`. See also`$g−>feature−available`.

$g−>xfs_growfs ($path [, datasec => $datasec] [, logsec => $logsec] [, rtsec => $rtsec] [, datasize => $datasize] [, logsize => $logsize] [, rtsize => $rtsize] [, rtextsize => $rtextsize] [, maxpct => $maxpct]);

Grow the XFS filesystem mounted at `path`.

The returned struct contains geometry information. Missing fields are returned as `−1` (for numeric fields) or empty string.

This function depends on the feature `xfs`. See also`$g−>feature−available`.

%info = $g−>xfs_info ($pathordevice);

`pathordevice` is a mounted XFS filesystem or a device containing an XFS filesystem. This command returns the geometry of the filesystem.

The returned struct contains geometry information. Missing fields are returned as `−1` (for numeric fields) or empty string.

This function depends on the feature `xfs`. See also`$g−>feature−available`.

$status = $g−>xfs_repair ($device [, forcelogzero => $forcelogzero] [, nomodify => $nomodify] [, noprefetch => $noprefetch] [, forcegeometry => $forcegeometry] [, maxmem => $maxmem] [, ihashsize => $ihashsize] [, bhashsize => $bhashsize] [, agstride => $agstride] [, logdev => $logdev] [, rtdev => $rtdev]);

Repair corrupt or damaged XFS filesystem on `device`.

The filesystem is specified using the `device` argument which should be the device name of the disk partition or volume containing the filesystem. If given the name of a block device, `xfs_repair` will attempt to find the raw device associated with the specified block device and will use the raw device instead.

Regardless, the filesystem to be repaired must be unmounted, otherwise, the resulting filesystem may be inconsistent or corrupt.

The returned status indicates whether filesystem corruption was detected (returns `1`) or was not detected (returns `0`).

This function depends on the feature `xfs`. See also`$g->feature-available`.

`$g`->yara_destroy ();

Destroy previously loaded Yara rules in order to free libguestfs resources.

This function depends on the feature `libyara`. See also`$g->feature-available`.

`$g`->yara_load ($filename);

Upload a set of Yara rules from local file *filename*.

Yara rules allow to categorize files based on textual or binary patterns within their content. See `$g->yara_scan` to see how to scan files with the loaded rules.

Rules can be in binary format, as when compiled with yarac command, or in source code format. In the latter case, the rules will be first compiled and then loaded.

Rules in source code format cannot include external files. In such cases, it is recommended to compile them first.

Previously loaded rules will be destroyed.

This function depends on the feature `libyara`. See also`$g->feature-available`.

`@detections` = `$g`->yara_scan ($path);

Scan a file with the previously loaded Yara rules.

For each matching rule, a `yara_detection` structure is returned.

The `yara_detection` structure contains the following fields.

`yara_name`

Path of the file matching a Yara rule.

`yara_rule`

Identifier of the Yara rule which matched against the given file.

This function depends on the feature `libyara`. See also`$g->feature-available`.

`@lines` = `$g`->zegrep ($regex, `$path`);

This calls the external `zegrep` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘grep’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`->zegrepi ($regex, `$path`);

This calls the external `zegrep -i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ‘‘PROTOCOL LIMITS’’ in **guestfs** (3).

This function is deprecated. In new code, use the ‘‘grep’’ call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$g`−>zero ($device);

This command writes zeroes over the first few blocks of `device`.

How many blocks are zeroed isn't specified (but itXs *not* enough to securely wipe the device). It should be sufficient to remove any partition tables, filesystem superblocks and so on.

If blocks are already zero, then this command avoids writing zeroes. This prevents the underlying device from becoming non-sparse or growing unnecessarily.

See also: `$g->zero_device`, `$g->scrub_device`, `$g->is_zero_device`

`$g`−>zero_device ($device);

This command writes zeroes over the entire `device`. Compare with`$g->zero` which just zeroes the first few blocks of a device.

If blocks are already zero, then this command avoids writing zeroes. This prevents the underlying device from becoming non-sparse or growing unnecessarily.

`$g`−>zero_free_space ($directory);

Zero the free space in the filesystem mounted on *directory*. The filesystem must be mounted read-write.

The filesystem contents are not affected, but any free space in the filesystem is freed.

Free space is not ''trimmed''. You may want to call `$g->fstrim` either as an alternative to this, or after calling this, depending on your requirements.

`$g`−>zerofree ($device);

This runs the *zerofree* program on `device`. This program claims to zero unused inodes and disk blocks on an ext2/3 filesystem, thus making it possible to compress the filesystem more effectively.

You should **not** run this program if the filesystem is mounted.

It is possible that using this program can damage the filesystem or data on the filesystem.

This function depends on the feature `zerofree`. See also`$g->feature-available`.

`@lines` = `$g`−>zfgrep ($pattern, `$path`);

This calls the external `zfgrep` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

This function is deprecated. In new code, use the ''grep'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`−>zfgrepi ($pattern, `$path`);

This calls the external `zfgrep -i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

This function is deprecated. In new code, use the ''grep'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`$description` = `$g`−>zfile ($meth, `$path`);

This command runs **file** (1) after first decompressing `path` using `meth`.

`meth` must be one of `gzip`, `compress` or `bzip2`.

Since 1.0.63, use `$g->file` instead which can now process compressed files.

This function is deprecated. In new code, use the ''file'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`−>zgrep ($regex, `$path`);

This calls the external **zgrep** (1) program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

This function is deprecated. In new code, use the ''grep'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

`@lines` = `$g`−>zgrepi ($regex, `$path`);

This calls the external `zgrep −i` program and returns the matching lines.

Because of the message protocol, there is a transfer limit of somewhere between 2MB and 4MB. See ''PROTOCOL LIMITS'' in **guestfs** (3).

This function is deprecated. In new code, use the ''grep'' call instead.

Deprecated functions will not be removed from the API, but the fact that they are deprecated indicates that there are problems with correct use of these functions.

AVAILABILITY

From time to time we add new libguestfs APIs. Also some libguestfs APIs won't be available in all builds of libguestfs (the Fedora build is full-featured, but other builds may disable features). How do you test whether the APIs that your Perl program needs are available in the version of `Sys::Guestfs` that you are using?

To test if a particular function is available in the `Sys::Guestfs` class, use the ordinary Perl UNIVERSAL method `can(METHOD)` (see **perlobj** (1)). For example:

```
use Sys::Guestfs;
if (defined (Sys::Guestfs−>can ("set_verbose"))) {
  print "\$g−>set_verbose is available\n";
}
```

To test if particular features are supported by the current build, use the ''feature_available'' method like the example below. Note that the appliance must be launched first.

```
$g−>feature_available ( ["augeas"] );
```

For further discussion on this topic, refer to ''AVAILABILITY'' in **guestfs** (3).

STORING DATA IN THE HANDLE

The handle returned from ''new'' is a hash reference. The hash normally contains some elements:

```
{
  _g => [private data used by libguestfs],
  _flags => [flags provided when creating the handle]
}
```

Callers can add other elements to this hash to store data for their own purposes. The data lasts for the lifetime of the handle.

Any fields whose names begin with an underscore are reserved for private use by libguestfs. We may add more in future.

It is recommended that callers prefix the name of their field(s) with some unique string, to avoid conflicts with other users.

COPYRIGHT

Copyright (C) 2009−2020 Red Hat Inc.

LICENSE

Please see the file COPYING.LIB for the full license.

SEE ALSO

guestfs (3), **guestfish** (1), <http://libguestfs.org>.