NAME

guestfs−performance − engineering libguestfs for greatest performance

DESCRIPTION

This page documents how to get the greatest performance out of libguestfs, especially when you expect to use libguestfs to manipulate thousands of virtual machines or disk images.

Three main areas are covered. Libguestfs runs an appliance (a small Linux distribution) inside qemu/KVM. The first two areas are: minimizing the time taken to start this appliance, and the number of times the appliance has to be started. The third area is shortening the time taken for inspection of VMs.

BASELINE MEASUREMENTS

Before making changes to how you use libguestfs, take baseline measurements.

Baseline: Starting the appliance

On an unloaded machine, time how long it takes to start up the appliance:

```
time guestfish −a /dev/null run
```

Run this command several times in a row and discard the first few runs, so that you are measuring a typical ‘‘hot cache’’ case.

Side note for developers: There is a program called *boot-benchmark* in https://github.com/libguestfs/libguestfs−analysis−tools which does the same thing, but performs multiple runs and prints the mean and standard deviation.

Explanation

The guestfish command above starts up the libguestfs appliance on a null disk, and then immediately shuts it down. The first time you run the command, it will create an appliance and cache it (usually under */var/tmp/.guestfs−**). Subsequent runs should reuse the cached appliance.

Expected results

You should expect to be getting times under 6 seconds. If the times you see on an unloaded machine are above this, then see the section ‘‘TROUBLESHOOTING POOR PERFORMANCE’’ below.

Baseline: Performing inspection of a guest

For this test you will need an unloaded machine and at least one real guest or disk image. If you are planning to use libguestfs against only X guests (eg. X = Windows), then using an X guest here would be most appropriate. If you are planning to run libguestfs against a mix of guests, then use a mix of guests for testing here.

Time how long it takes to perform inspection and mount the disks of the guest. Use the first command if you will be using disk images, and the second command if you will be using libvirt.

```
time guestfish −−ro −a disk.img −i exit

time guestfish −−ro −d GuestName −i exit
```

Run the command several times in a row and discard the first few runs, so that you are measuring a typical ‘‘hot cache’’ case.

Explanation

This command starts up the libguestfs appliance on the named disk image or libvirt guest, performs libguestfs inspection on it (see ‘‘INSPECTION’’ in **guestfs**(3)), mounts the guest’s disks, then discards all these results and shuts down.

The first time you run the command, it will create an appliance and cache it (usually under */var/tmp/.guestfs−**). Subsequent runs should reuse the cached appliance.

Expected results

You should expect times which are ≤ 5 seconds greater than measured in the first baseline test above. (For example, if the first baseline test ran in 5 seconds, then this test should run in ≤ 10 seconds).

UNDERSTANDING THE APPLIANCE AND WHEN IT IS BUILT/CACHED

The first time you use libguestfs, it will build and cache an appliance. This is usually in */var/tmp/.guestfs−**, unless you have set `$TMPDIR` or `$LIBGUESTFS_CACHEDIR` in which case it will be under that temporary directory.

For more information about how the appliance is constructed, see "SUPERMIN APPLIANCES" in **supermin** (1).

Every time libguestfs runs it will check that no host files used by the appliance have changed. If any have, then the appliance is rebuilt. This usually happens when a package is installed or updated on the host (eg. using programs like `yum` or `apt−get`). The reason for reconstructing the appliance is security: the new program that has been installed might contain a security fix, and so we want to include the fixed program in the appliance automatically.

These are the performance implications:

- The process of building (or rebuilding) the cached appliance is slow, and you can avoid this happening by using a fixed appliance (see below).
- If not using a fixed appliance, be aware that updating software on the host will cause a one time rebuild of the appliance.
- */var/tmp* (or `$TMPDIR`, `$LIBGUESTFS_CACHEDIR`) should be on a fast disk, and have plenty of space for the appliance.

USING A FIXED APPLIANCE

To fully control when the appliance is built, you can build a fixed appliance. This appliance should be stored on a fast local disk.

To build the appliance, run the command:

```
 libguestfs−make−fixed−appliance <directory>
```

replacing `<directory>` with the name of a directory where the appliance will be stored (normally you would name a subdirectory, for example: */usr/local/lib/guestfs/appliance* or */dev/shm/appliance*).

Then set `$LIBGUESTFS_PATH` (and ensure this environment variable is set in your libguestfs program), or modify your program so it calls `guestfs_set_path`. For example:

```
 export LIBGUESTFS_PATH=/usr/local/lib/guestfs/appliance
```

Now you can run libguestfs programs, virt tools, guestfish etc. as normal. The programs will use your fixed appliance, and will not ever build, rebuild, or cache their own appliance.

(For detailed information on this subject, see: **libguestfs−make−fixed−appliance** (1)).

Performance of the fixed appliance

In our testing we did not find that using a fixed appliance gave any measurable performance benefit, even when the appliance was located in memory (ie. on */dev/shm*). However there are two points to consider:

1. Using a fixed appliance stops libguestfs from ever rebuilding the appliance, meaning that libguestfs will have more predictable start-up times.
2. The appliance is loaded on demand. A simple test such as:

   ```
    time guestfish −a /dev/null run
   ```

   does not load very much of the appliance. A real libguestfs program using complicated API calls would demand-load a lot more of the appliance. Being able to store the appliance in a specified location makes the performance more predictable.

REDUCING THE NUMBER OF TIMES THE APPLIANCE IS LAUNCHED

By far the most effective, though not always the simplest way to get good performance is to ensure that the appliance is launched the minimum number of times. This will probably involve changing your libguestfs application.

Try to call `guestfs_launch` at most once per target virtual machine or disk image.

Instead of using a separate instance of **guestfish**(1) to make a series of changes to the same guest, use a single instance of guestfish and/or use the guestfish −−*listen* option.

Consider writing your program as a daemon which holds a guest open while making a series of changes. Or marshal all the operations you want to perform before opening the guest.

You can also try adding disks from multiple guests to a single appliance. Before trying this, note the following points:

1. Adding multiple guests to one appliance is a security problem because it may allow one guest to interfere with the disks of another guest. Only do it if you trust all the guests, or if you can group guests by trust.
2. There is a hard limit to the number of disks you can add to a single appliance. Call ‘‘guestfs_max_disks’’ in **guestfs**(3) to get this limit. For further information see ‘‘LIMITS’’ in **guestfs**(3).
3. Using libguestfs this way is complicated. Disks can have unexpected interactions: for example, if two guests use the same UUID for a filesystem (because they were cloned), or have volume groups with the same name (but see `guestfs_lvm_set_filter`).

virt−df(1) adds multiple disks by default, so the source code for this program would be a good place to start.

SHORTENING THE TIME TAKEN FOR INSPECTION OF VMs

The main advice is obvious: Do not perform inspection (which is expensive) unless you need the results.

If you previously performed inspection on the guest, then it may be safe to cache and reuse the results from last time.

Some disks don’t need to be inspected at all: for example, if you are creating a disk image, or if the disk image is not a VM, or if the disk image has a known layout.

Even when basic inspection (`guestfs_inspect_os`) is required, auxiliary inspection operations may be avoided:

- Mounting disks is only necessary to get further filesystem information.
- Listing applications (`guestfs_inspect_list_applications`) is an expensive operation on Linux, but almost free on Windows.
- Generating a guest icon (`guestfs_inspect_get_icon`) is cheap on Linux but expensive on Windows.

PARALLEL APPLIANCES

Libguestfs appliances are mostly I/O bound and you can launch multiple appliances in parallel. Provided there is enough free memory, there should be little difference in launching 1 appliance vs N appliances in parallel.

On a 2−core (4−thread) laptop with 16 GB of RAM, using the (not especially realistic) test Perl script below, the following plot shows excellent scalability when running between 1 and 20 appliances in parallel:

```
 12 ++---+----+----+----+-----+----+----+----+----+---++
    +    +    +    +    +     +    +    +    +    +    *
    |                                                  |
    |                                               *  |
 11 ++                                                ++
    |                                                  |
    |                                                  |
    |                                          *  *    |
 10 ++                                                ++
    |                                        *         |
    |                                                  |
s   |                                                  |
  9 ++                                                ++
e   |                                                  |
    |                                     *            |
c   |                                                  |
  8 ++                                  *             ++
o   |                                *                 |
    |                                                  |
n 7 ++                                                ++
    |                              *                   |
d   |                           *                      |
    |                                                  |
s 6 ++                                                ++
    |                      *  *                        |
    |                   *                              |
    |                                                  |
  5 ++                                                ++
    |                                                  |
    |                 *                                |
    |            * *                                   |
  4 ++                                                ++
    |                                                  |
    |                                                  |
    +    *  * *    +    +     +    +    +    +    +    +
  3 ++-*-+----+----+----+-----+----+----+----+----+---++
    0    2    4    6    8     10   12   14   16   18   20
              number of parallel appliances
```

It is possible to run many more than 20 appliances in parallel, but if you are using the libvirt backend then you should be aware that out of the box libvirt limits the number of client connections to 20.

The simple Perl script below was used to collect the data for the plot above, but there is much more information on this subject, including more advanced test scripts and graphs, available in the following blog postings:

http://rwmj.wordpress.com/2013/02/25/multiple−libguestfs−appliances−in−parallel−part−1/
http://rwmj.wordpress.com/2013/02/25/multiple−libguestfs−appliances−in−parallel−part−2/
http://rwmj.wordpress.com/2013/02/25/multiple−libguestfs−appliances−in−parallel−part−3/
http://rwmj.wordpress.com/2013/02/25/multiple−libguestfs−appliances−in−parallel−part−4/

```
#!/usr/bin/env perl

use strict;
use threads;
use warnings;
use Sys::Guestfs;
use Time::HiRes qw(time);

sub test {
    my $g = Sys::Guestfs−>new;
    $g−>add_drive_ro ("/dev/null");
    $g−>launch ();

    # You could add some work for libguestfs to do here.

    $g−>close ();
}
```

```
 # Get everything into cache.
 test (); test (); test ();

 for my $nr_threads (1..20) {
     my $start_t = time ();
     my @threads;
     foreach (1..$nr_threads) {
         push @threads, threads->create (\&test)
     }
     foreach (@threads) {
         $_->join ();
         if (my $err = $_->error ()) {
             die "launch failed with $nr_threads threads: $err"
         }
     }
     my $end_t = time ();
     printf ("%d %.2f\n", $nr_threads, $end_t - $start_t);
 }
```

USING USER-MODE LINUX

Since libguestfs 1.24, it has been possible to use the User-Mode Linux (uml) backend instead of KVM (see "USER-MODE LINUX BACKEND" in **guestfs**(3)). This section makes some general remarks about this backend, but it is **highly advisable** to measure your own workload under UML rather than trusting comments or intuition.

- UML usually performs the same or slightly slower than KVM, on baremetal.
- However UML often performs the same under virtualization as it does on baremetal, whereas KVM can run much slower under virtualization (since hardware virt acceleration is not available).
- Upload and download is as much as 10 times slower on UML than KVM. Libguestfs sends this data over the UML emulated serial port, which is far less efficient than KVM's virtio-serial.
- UML lacks some features (eg. qcow2 support), so it may not be applicable at all.

For some actual figures, see: http://rwmj.wordpress.com/2013/08/14/performance-of-user-mode-linux-as-a-libguestfs-backend/#content

TROUBLESHOOTING POOR PERFORMANCE

Ensure hardware virtualization is available

Use */proc/cpuinfo* to ensure that hardware virtualization is available. Note that you may need to enable it in your BIOS.

Hardware virt is not usually available inside VMs, and libguestfs will run slowly inside another virtual machine whatever you do. Nested virtualization does not work well in our experience, and is certainly no substitute for running libguestfs on baremetal.

Ensure KVM is available

Ensure that KVM is enabled and available to the user that will run libguestfs. It should be safe to set 0666 permissions on */dev/kvm* and most distributions now do this.

Processors to avoid

Avoid processors that don't have hardware virtualization, and some processors which are simply very slow (AMD Geode being a great example).

Xen dom0

In Xen, dom0 is a virtual machine, and so hardware virtualization is not available.

Use libguestfs ≥ 1.34 and qemu ≥ 2.7

During the libguestfs 1.33 development cycle, we spent a large amount of time concentrating on boot performance, and added some patches to libguestfs, qemu and Linux which in some cases can reduce boot times to well under 1 second. You may therefore get much better performance by moving to the versions of libguestfs or qemu mentioned in the heading.

DETAILED ANALYSIS

Boot analysis

In https://github.com/libguestfs/libguestfs−analysis−tools is a program called `boot−analysis`. This program is able to produce a very detailed breakdown of the boot steps (eg. qemu, BIOS, kernel, libguestfs init script), and can measure how long it takes to perform each step.

Detailed timings using ts

Use the **ts** (1) command (from moreutils) to show detailed timings:

```
$ guestfish −a /dev/null run −v |& ts −i '%.s'
0.000022 libguestfs: launch: program=guestfish
0.000134 libguestfs: launch: version=1.29.31fedora=23,release=2.fc23,libvirt
0.000044 libguestfs: launch: backend registered: unix
0.000035 libguestfs: launch: backend registered: uml
0.000035 libguestfs: launch: backend registered: libvirt
0.000032 libguestfs: launch: backend registered: direct
0.000030 libguestfs: launch: backend=libvirt
0.000031 libguestfs: launch: tmpdir=/tmp/libguestfsw18rBQ
0.000029 libguestfs: launch: umask=0002
0.000031 libguestfs: launch: euid=1000
0.000030 libguestfs: libvirt version = 1002012 (1.2.12)
[etc]
```

The timestamps are seconds (incrementally since the previous line).

Detailed timings using SystemTap

You can use SystemTap (**stap** (1)) to get detailed timings from libguestfs programs.

Save the following script as *time.stap*:

```
global last;

function display_time () {
      now = gettimeofday_us ();
      delta = 0;
      if (last > 0)
            delta = now − last;
      last = now;

      printf ("%d (+%d):", now, delta);
}

probe begin {
      last = 0;
      printf ("ready\n");
}

/* Display all calls to static markers. */
probe process("/usr/lib*/libguestfs.so.0")
          .provider("guestfs").mark("*") ? {
      display_time();
      printf ("\t%s %s\n", $$name, $$parms);
}

/* Display all calls to guestfs_* functions. */
probe process("/usr/lib*/libguestfs.so.0")
          .function("guestfs_[a−z]*") ? {
```

```
        display_time();
        printf ("\t%s %s\n", probefunc(), $$parms);
  }
```

Run it as root in one window:

```
# stap time.stap
ready
```

It prints "ready" when SystemTap has loaded the program. Run your libguestfs program, guestfish or a virt tool in another window. For example:

```
$ guestfish -a /dev/null run
```

In the stap window you will see a large amount of output, with the time taken for each step shown (microseconds in parenthesis). For example:

```
xxxx (+0):     guestfs_create
xxxx (+29):    guestfs_set_pgroup g=0x17a9de0 pgroup=0x1
xxxx (+9):     guestfs_add_drive_opts_argv g=0x17a9de0 [...]
xxxx (+8):     guestfs_int_safe_strdup g=0x17a9de0 str=0x7f8a153bed5d
xxxx (+19):    guestfs_int_safe_malloc g=0x17a9de0 nbytes=0x38
xxxx (+5):     guestfs_int_safe_strdup g=0x17a9de0 str=0x17a9f60
xxxx (+10):    guestfs_launch g=0x17a9de0
xxxx (+4):     launch_start
[etc]
```

You will need to consult, and even modify, the source to libguestfs to fully understand the output.

Detailed debugging using gdb

You can attach to the appliance BIOS/kernel using gdb. If you know what you're doing, this can be a useful way to diagnose boot regressions.

Firstly, you have to change qemu so it runs with the `-S` and `-s` options. These options cause qemu to pause at boot and allow you to attach a debugger. Read**qemu**(1) for further information. Libguestfs invokes qemu several times (to scan the help output and so on) and you only want the final invocation of qemu to use these options, so use a qemu wrapper script like this:

```
#!/bin/bash -

# Set this to point to the real qemu binary.
qemu=/usr/bin/qemu-kvm

if [ "$1" != "-global" ]; then
    # Scanning help output etc.
    exec $qemu "$@"
else
    # Really running qemu.
    exec $qemu -S -s "$@"
fi
```

Now run guestfish or another libguestfs tool with the qemu wrapper (see "QEMU WRAPPERS" in **guestfs**(3) to understand what this is doing):

```
LIBGUESTFS_HV=/path/to/qemu-wrapper guestfish -a /dev/null -v run
```

This should pause just after qemu launches. In another window, attach to qemu using gdb:

```
$ gdb
(gdb) set architecture i8086
The target architecture is assumed to be i8086
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) cont
```

At this point you can use standard gdb techniques, eg. hitting `^C` to interrupt the boot and `bt` get a stack trace, setting breakpoints, etc. Note that when you are past the BIOS and into the Linux kernel, you'll want to change the architecture back to 32 or 64 bit.

PERFORMANCE REGRESSIONS IN OTHER PROGRAMS

Sometimes performance regressions happen in other programs (eg. qemu, the kernel) that cause problems for libguestfs.

In https://github.com/libguestfs/libguestfs−analysis−tools *boot−benchmark/boot−benchmark−range.pl* is a script which can be used to benchmark libguestfs across a range of git commits in another project to find out if any commit is causing a slowdown (or speedup).

To find out how to use this script, consult the manual:

```
./boot−benchmark/boot−benchmark−range.pl −−man
```

SEE ALSO

supermin (1), **guestfish** (1), **guestfs** (3), **guestfs−examples** (3), **guestfs−internals** (1), **libguestfs−make−fixed−appliance** (1), **stap** (1), **qemu** (1), **gdb** (1), http://libguestfs.org/.

AUTHORS

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

COPYRIGHT

LICENSE

BUGS

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

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

When reporting a bug, please supply:

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