Virt-backup utility

Design document

Draft v0.0.1

This document describes the fundamentals of virt-backup utility which should be the utility to backup libvirt guests. It also presents multithreaded compression system to optimize compression speed. This libvirt-based tool enables the user to save his/her guests in in the compressed format to save file. Virt-backup has option to use LZMA compression or disable the compression entirely however the compression done by this algorithm is pretty slow so it needs to be optimized for better speed and performance. This design document also covers the options to implement the compression in multiple threads using the communication protocol to split and process the input and output data.

Document overview

1. Virt-backup tool introduction

2. Multi-Threaded Compression Protocol (MTC Protocol or MTCP)

3. Compression and decompression implementation

1. Virt-backup tool introduction

This paper introduces the virt-backup tool which should be the guest backup/archiver tool allowing user to save his/her guests both in the standard (uncompressed) and also compressed format. There is also one issue we have to face which is procedure to backup a running guest.

We have to keep in mind that guest disk images could be backed-up by both files and partitions. According to the direct file/device read and write we could perform those operations on both files and partitions however this approach seems to be good only for non-running guests. To prevent inconsistency of the images used by running guests we should implement a pause code or code similar to the live migration to make sure the guest will not be using the image for the time the backup will be in progress and therefore the guest image will remain consistent.

To make sure the disk image and the guest memory will be consistent after user restores the guest from it's backup copies we should save the guest memory and backup the image at the very same time. This should be easily done by calling virDomainSave() followed by virDomainGetXMLDesc() call to get the domain XML and parse disk devices for the domain to be saved. This approach has the disadvantage of having to shutdown the guest which is done automatically after virDomainSave() call. Unfortunately there is no libvirt flag API to do the live save (or something similar to live migration).

If we consider doing backup for inactive (not running) guests the procedure is easy – all we have to do is just to get the XML description for the domain, parse them to get the disk images storage locations and then run the compression or copy on them.

The command syntax should be as follows:

```
virt-backup backup --destination-path path [--hypervisor-uri uri] [--domain-names names] [--compression type]
```

to do the backup or

```
virt-backup restore --source-path src --destination-path path [--hypervisor-uri uri] [--allow-autorun] [--domain-names names]
```

to restore the guests from the source path src and save to the new location to the destination path. The size information for the original images will be saved in the XML file on the backup path. There will be several information saved including the originating domain, type, original file name, original file size, compression used (none or LZMA), also file ownership and SELinux context if applicable and also identifier whether the domain was originally running or not. If the domain was running then there should be the file called <domainName>.sav having all the disk memory saved in it. In all cases there should be a <domainName>.xml file present having the domain's libvirt XML description scheme to be restored.

The restore code of the utility will be reading the state information from the XML file and if the domain memory has been saved (i.e. the domain was running before the process) then all the guest memory will be restored after all disk images are restored to startup the guest immediately after the process if autorun flag is allowed using the “--allow-autorun” option.

You can also pass additional “--domain-names” syntax followed by the domain name(s) to identify the list of the domains to be saved or restored ignoring the rest of the domains found.

2. Multi-Threaded Compression Protocol

Since the compression is the main key why to implement such an utility in the form of the standalone application rather than bash script using virsh we need to optimize the compression speed. The compression speed for LZMA compression is rather slow however the compression ratio is very high and therefore it saves as much space as possible. For this project the Multi-Threaded Compression Protocol (MTCP for short) was designed to specify exactly the needs of separating the data between multiple processes.

The MTCP design itself presents the infrastructure of one main process called the control process that's being forked into many child processes doing the compression job itself.

The control process doesn't do any compression itself and it's task to send original data to child processes and receive the compressed data from child processes. To identify which child is working on what chunk the control process have to keep track on child process ids and assign the chunk to every child. The chunk have to be identified by it's identifier called the chunk_id which could serve as a source for chunk position computations since the buffer size will be a constant variable for entire run. There the formula could be that easy: $chunk_{id}$ = <(id * chunk size), ((id + 1) * chunk size) – 1> which means that if you have e.g. third chunk (id = 3) for chunk size of 1024 Megabytes (i.e. 1073741824 bytes) then the chunk will start on position 3072 Megabytes (3 * 1073741824, i.e. 3221225472 bytes) and it will end on the position of 4096 Megabytes without one byte (((3 + 1) * 1073741824) – 1, i.e. 4294967295 bytes) and therefore it could be written as $chunk_{id}$ = <3221225472, 4294967295>.

All the child processes have to use the appropriate amount of memory not to run system into the panic (OOM killer) and this should be controlled by the control process too. All the child processes should be using the same amount of memory and therefore it should be possible to identify the amount of overall memory available for the compression, i.e. for all the child processes. Since all of the compression levels used by LZMA are having different buffer sizes like e.g. 6 MiB buffer size for compression levels 0 and 1, 60 MiB buffer size for compression level 5 and 800 MiB buffer size for compression level 9 then the number of child processes could be calculated as:

child_count = ((total_memory – 256 MiB) / buffer size for compression level)

For example, on 2 GiB (1957 MiB) system you'll get the value of 1957 – 256 = 1701 so for the compression level 5 (buffer size 60 MiB) there will be 28 (1701 / 60) child processes spawned and for the compression level 9 (buffer size 800 MiB) there will be just 2 processes spawned (1701 / 800). It may be good to add option to allow manual override of the buffer size per compression level which would also require adding the check not to allow system to run out of memory and trigger OOM killer.

The control process will be communicating with child processes using the communication protocol:

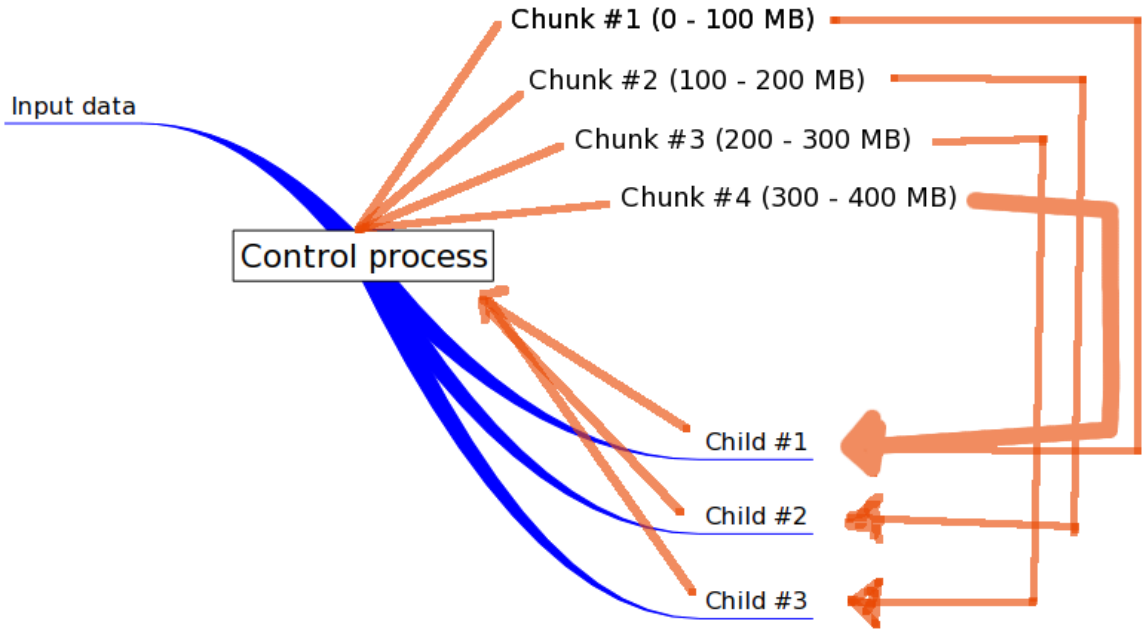
1. Control process creates the child processes
2. Each child process send an idle signal (“{CHILDPID}I”) once it's created
3. Control process assigns one chunk to each idle child using syntax like (“{CHILDPID}C{CHUNKID}S{SIZE}{DATA}) where SIZE is encoded on 4-bytes (32-bits)
4. Once the child finishes is tells control process using “{CHILDPID}D{CHUNKID}” where D stands for “done” word.
5. Control process keeps track of all flushed buffers and if this is the next id then it reads it from the child using “{CHILDPID}F{CHUNKID}”, child process replies using the protocol as

"{CHILDPID}C{CHUNKID}S{SIZE}{DATA}". After this step child frees the buffer and sends the idle signal again.

List of commands in the protocol:

1. Sending idle child identification - "{CHILDPID}I", sent by child
2. Sending data identification - "{CHILDPID}C{CHUNKID}S{SIZE}{DATA}"
3. Sending done child identification - "{CHILDPID}D{CHUNKID}", sent by child
4. Sending flush (read from child) request - "{CHILDPID}F{CHUNKID}"
5. Sending quit request to the child - "{CHILDPID}Q"

The communication diagram is simple:



Step 1: Control process splits the data into chunks, each chunk is being sent to the idle child (child sends "{CHILDPID}I" to identify it's idle)
Step 2: Control process is seeing the idle child so it sends a chunk to the child (identified by it's PID) using "{CHILDPID}C{CHUNKID}S{SIZE}{DATA}" where SIZE is encoded on 4-bytes (32-bits)
Step 3: Child finishes the work on the chunk so it's ready to send the data back to control proc. so it posts the acknowledge it's ready using "{CHILDPID}D{CHUNKID}".
Step 4: Control process is seeing the child is ready so it looks for the last flushed buffer and if the buffer to be output is having this ID then it posts "{CHILDPID}F{CHUNKID}" to the child otherwise it waits.

This is being repeated until there are still some input data in the file opened by the control process. If it's there are no data available in the input file then "{CHILDPID}Q" command should be send to all the child processes that are in the idle state since there are no new data they could work on.

3. Compression and decompression implementation

The compression should be implemented using the liblzma library that supports the multi-threaded usage so using it for multi-threaded compression as described in chapter 2 (Multi-Threaded Compression Protocol) should be working fine with the liblzma's implementation of LZMA compression.

The compression will be having the buffer sizes preset according to the xz command man page since this page contains useful information on the buffer sizes being used by this tool however the option to manually override the buffer size for chosen compressed level should be added with warning that wrong setting of compression buffer sizes may result into ineffective compression.

The decompression should be done by allocating a big buffer in memory and trying to decompress everything normally since decompression is pretty fast for LZMA algorithm. Just the compression speed should matter there since when user supplies high compression level then the compression should be very slow however this shouldn't be the decompression issue.