CERN - CONSEIL EUROPEN POUR LA RECHERCHE NUCLAIRE
PH-SFT DEPARTMENT
CH-1211 GENEVA 23, SWITZERLAND

Petra Mazdin

Optimization of μCernVM early boot process

Supervised by: Jakob Blomer Gerardo Ganis

October 2015

1 Abstract

CernVM virtual machine is a Linux based virtual appliance optimized for High Energy Physics experiments. It is used for cloud computing, volunteer computing, and software development by the four large LHC experiments. The goal of this project is profiling and optimizing the boot process of the CernVM. A key part was the development of a performance profiler for shell scripts as an extension to the popular BusyBox open source UNIX tool suite. Based on the measurements, costly shell code was replaced by more efficient, custom C programs. The results are compared to the original ones and successful optimization is proven.

2 Introduction

The aim of this project is to optimize μCernVM early boot process, i.e. to minimize execution time of its steps. Since this virtual machine is widely used by lots of physicists participating in LHC experiments, located at CERN or around the world, desired optimization would have great significance. In order to get duration of booting steps, BusyBox ash was used as an interpreter of shell scripts that were executing in the early phase of machine booting. Since the original version of BusyBox was not suitable to provide all necessary data, modifications were applied as it is described in the section 3.3.
The results of running a machine using this modified tool, both for a first boot and reboot, are given in the section 3.4. Afterwards, when having a list of steps whose execution time was too long, optimization explained in the section 4 was applied. Demanding parts of code were replaced by C programs whose purpose was to run different processes in parallel. Finally, the results for a total time of this booting phase showed improvement and are given in the section 5, together with the other possible modifications.

3 Process description

3.1 The μCernVM Virtual Machine

CernVM virtual machine [3] is a Linux based Virtual Software Appliance for the participants of CERN LHC experiments. The Appliance represents a complete, portable and easy to configure user environment for developing and running LHC data analysis locally and on institutional and commercial clouds, independently of Operating System software and hardware platform. The goal is to remove a need for the installation of the experiment software and to minimize the number of platforms on which experiment software needs to be supported and tested.

Motivation to use this virtual machine and desire to optimize it can be shown by a number of boots per month. This statistics is provided by web server logs where virtual machines call on boot and are given for the time period starting in January 2015 until September 2015. There was

- 175,000 fresh boots per month
- 1 million reboots of existing VMs per month

The μCernVM image described in [1] consists of a Linux kernel and an init ramdisk that includes the CernVM-FS client as it is shown in the Figure 1. The Linux kernel is slim as it only requires device drivers for the handful of hypervisors around. Furthermore, the μCernVM kernel contains the union file system AUFS which is needed because CernVM-FS is a read-only file system.

Local changes, such as of files in /etc or /var, cannot be written to CernVM-FS. Instead, the union file system transparently redirects such changes to a locally attached scratch hard disk. The local hard disk is also used to store the CernVM-FS cache. The init ramdisk contains the CernVM-FS client and a steering script whose purpose is explained in the next section.

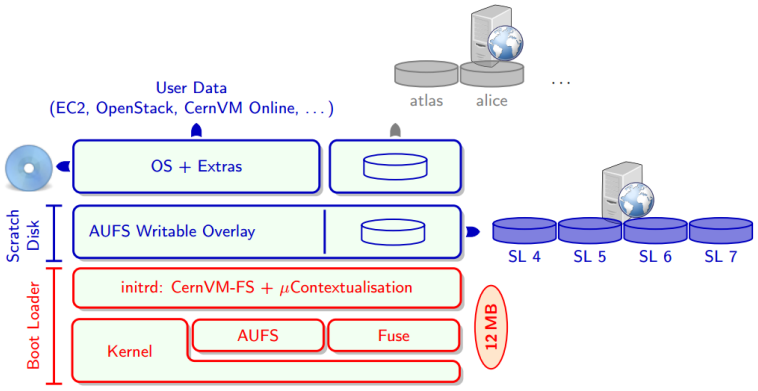

Figure 1: Building blocks of CernVM

3.2 Steps of the early boot process

At the beginning of the Linux boot process described in [2], in the so called early user space, machine starts networking and tries to connect to Network Time Protocol server in order to get current time. The Linux kernel uses a μCernVM root file system in memory provided by the init ramdisk. This is a part where the necessary storage device drivers are loaded to access the actual root file system.

The purpose of the steering script is to create the virtual machine's root file system stack that is constructed by unifying the CernVM-FS mount point with the writable scratch space. To do so, the steering script can process contextualization information from various sources, such as OpenStack, OpenNebula, or Amazon EC2. Based on the contextualization information, the CernVM-FS repository and the repository version is selected. The amount of data that needs to be loaded in order to boot the virtual machine sums up to 12 MB for the image and an additional 100 MB downloaded by the CernVM-FS client in order to boot Scientific Linux 6 from CernVM-FS. The CernVM-FS infrastructure used to distribute experiment software can be reused so that a booting μCernVM virtual machine starts practically instantaneously.

Once the actual root file system is available, the system switches its root file system to the new root mount point after which the previous root file system becomes useless and is removed from memory. First, the scratch hard disk is mounted on /root.rw. μCernVM grabs the first empty hard disk or partition attached to the virtual machine, or remaining free space on the

boot hard disk. It automatically partitions, formats, and labels the scratch space. Due to the file system label, μCernVM finds an already prepared scratch space on next boot. The scratch space is used as a persistent writable overlay for local changes to the root file system and as a cache for the CernVM-FS client that loads the operating system. Secondly, a CernVM-FS repository containing a template operating system installation is mounted on /root.ro. The file system unification of the CernVM-FS mount point and the directory containing the persistent overlay is mounted on /root.

Finally, the /root directory is installed as a new root file system. The /root.ro and /root.rw directories are projected into the final root file system via bind mounts.

Moreover, essential system files need to be pinned in the CernVM-FS cache to be able to change root file system. When finished, the rest of the init ramdisk is removed from memory and a real operating system can start.

3.3 BusyBox

BusyBox [6] is a single binary, which is a conglomerate of many applications, each of which can be accessed by calling the single BusyBox binary with various names in a specific manner with appropriate arguments.

BusyBox can be customized to provide a subset of over two hundred utilities. It can provide most of the utilities specified in the Single Unix Specification plus many others that a user would expect to see on a Linux system. It uses the Almquist shell, also known as A Shell, ash and sh.

BusyBox benefits from the single binary approach, as it reduces the overhead introduced by the executable file format, and it allows code to be shared between multiple applications without requiring a library. BusyBox provides simplified versions of the utilities, e.g. an ls command without file sorting ability.

Since it represents a tool to inteprete shell scripts and all the earlier mentioned steps of the early boot process of the μCernVM are actually executing in bash scripts, BusyBox proved to be a good profiling tool to measure execution time of each step in this booting phase.

3.3.1 Modifications of BusyBox

In order to get execution time of commands, original version of BusyBox asks for a modification.

The idea is to take start and end time separately for each executed step. In order to avoid wrong interpretation between different commands, each time pid (process ID), start/end time and exact command are taken so that it was possible to analyze them by the same pid and calculate duration. All data are then saved in a list whose elements differ by its type: start pid (long int), end pid (long int), start time (long int), end time (long int) and command (char *).

Before starting any modification, a process needs to create a copy of itself so that changes can be applied. When creating a new process, forking the current process is necessary, i.e. a new child process which is exactly the same as the parent process is created. After that, the child process does an exec system call to replace all the data from the parent process with the modified ones.

Forking and hotspots that need to be modified are shown in Figure 2.

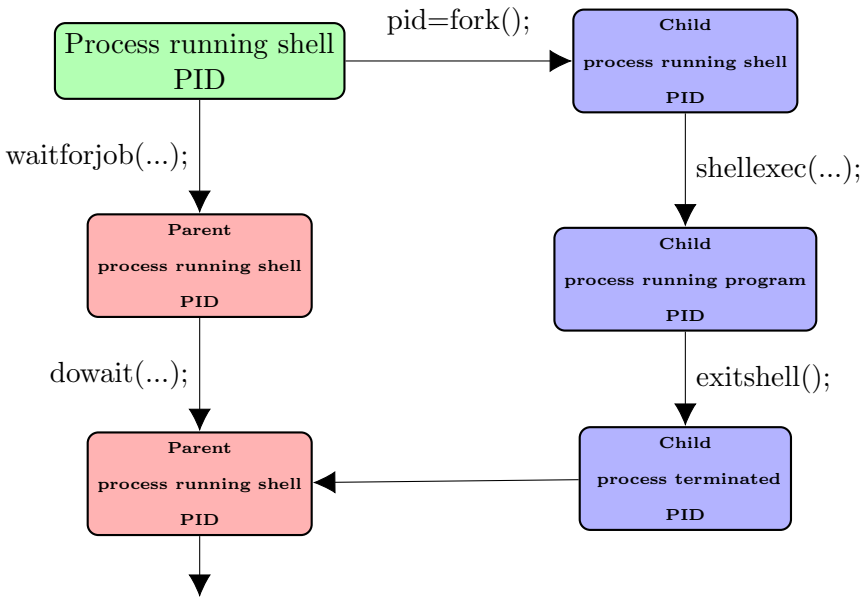

Figure 2: Hotspots of BusyBox ash to be modified

A part of BusyBox ash where start time should be taken is right before dowait() where parent process starts to wait for forked child process to finish

its execution (all in waitforjob()). When ps ps_status becomes 1, meaning that a child is finished, exact time stamp is taken as end time.

Moreover, pipelined processes ask for a special treatment, meaning that it can happen that there exist some processes executing in parallel, as it is shown in Figure 3. What happens is that final time stamps that are proceeded to further analysis are the ones taken at the moment the last part of a pipeline is executed. That is, of course, wrong because parts are executed in parallel and any on them can finish earlier.

```
echo -n "pin ${file}" | nc local:/${ROOT}/mnt/.rw/cache/${CVMFS_REPOS}/cvmfs_io.${CVMFS_REPOS}!14407541947078091529
!!1529!1440754194707815

echo -n "pin ${file}"!14407541947102991534
nc local:/${ROOT}/mnt/.rw/cache/${CVMFS_REPOS}/cvmfs_io.${CVMFS_REPOS}!14407541947103001535
!!1534!-1
!!1535!1440754194712287

echo -n "pin ${file}"!14407541947122871534
nc local:/${ROOT}/mnt/.rw/cache/${CVMFS_REPOS}/cvmfs_io.${CVMFS_REPOS}!-1!1535
!!1534!1440754194712395
!!1535!1440754194712396
```

Figure 3: Example of a set of processes that are part of a pipeline

In order to avoid that, ps ps_status is checked every time. If a process is not finished, the start time is taken and if the process is finished, the end time is taken. For anything else, '-1' is written as a time stamp. This way it makes it easier to analyze data later to take the lowest time stamps and avoid '-1'-s.

Furthermore, not to lose obtained data, they must be saved. That is why, each time when exiting a shell, in exitshell(), a new .csv file is created and saved to a path given by the environment variable ASH_SCORES.

When BusyBox ash intepreted everything it had to, all data are contained in a certain number of .csv files that can be analyzed using two python files to get final results that include command name, total execution time because some of them were repeated, number of repetitions and average execution time.

All these modifications are going to be applied only if it is defined by #define GET_TIME_CONSUMPTION in the beginning.

3.4 Detecting Hot Spots

When having modified BusyBox ash, steps that need to be executed in order to get results that can be analyzed are:

- export ASH_SCORES=/path/to/directory (to create an environment variable to save all results)
- interprete shell scripts using BusyBox
- run main.py that invokes a method from collect_data.py to analyze time stamps; final result is saved to 'sortedAll' in the earlier created ASH_SCORES

When talking about virtual machine, there are two possibilites to get results from: a first boot and a reboot since many steps are skipped in the following boots, e.g. downloading user_data, partitioning scratch device, etc. Therefore, it is expected to get less executed commands in the 'reboot' results.

3.4.1 First Boot

Command	$\sum$ [ms]	#repeats	AVG [ms]
nc -w ${UCONTEXT_TIMEOUT} ${server}...	8020	4	2005
modprobe -q floppy	6051	1	6051
timeout -t 5 ntpclient -s -h ${NTP_PEER}...	2132	1	2132
nc local:/${ROOT}/mnt/.rw/cache...	1294	154	8
ls ${AUFS_RO}...	1218	1	1218
echo -n "pin ${file}"	729	154	4
TOTAL DURATION:			~25.21 s
# DIFFERENT COMMANDS:			155
# COMMANDS:			1135

Table 1: Execution times for a first boot

Data shown in Table 1 are only hotspots, i.e. commands that need the most time to be executed. As it is shown, whole booting phase lasts cca 25 seconds, where 155 different commands are executed and since some of them are repeated, 1135 commands are executed in total. To start with optimization, these data need to be analyzed because not all of commands can

be changed to the ones that take less time. The first highlighted command 'nc -w ${UCONTEXT_TIMEOUT} ${server}...' is related to connecting to different servers and can be optimized as you will see in section 4.

Concerning the second command, the floppy module could not be loaded by default. In such case, it could be loaded with : 'modprobe -q floppy' and that can not be changed. Further commands can be optimized because they are related to obtaining time information using Network Time Protocol that takes too much time and pinning system files to the cache that stops the booting phase and only continues when it finishes.

3.4.2 Reboot

Command	$\sum$ [ms]	#repeats	AVG [ms]
modprobe -q floppy	6057	1	6057
timeout -t 5 ntpclient -s -h ${NTP_PEER}...	1475	1	1475
nc local:/${ROOT}/mnt/.rw/cache...	1232	154	8
echo -n "pin ${file}"	608	154	4
TOTAL DURATION:			∼13.32 s
# DIFFERENT COMMANDS:			128
# COMMANDS:			758

Table 2: Execution times for a reboot

As in previous case, Table 2 shows hotspots, but this time of a reboot. Obviously, a number of executed steps is smaller, 128 different commands, and 758 in total. Whole booting lasts cca 13 seconds, almost half of the execution time of the first boot. There is no more command related to downloading user_data since once it is done, it doesn't have to download again. But those commands regarding to the time and pinning files still exist and also represent hotspots, i.e. a part to be optimized.

4 Optimization

As it was mentioned in the previous section, parts that showed a need for optimization were related to downloading user_data, getting time from NTP server and pinning system files. Not all of them were optimized in this project, but they should be considered in the future.

4.1 Contextualization

A context is a small (up to 16kB), usually human-readable snippet that is used to apply a role to a virtual machine. A context allows to have a singe virtual machine image that can back many different virtual machine instances in so far as the instance can adapt to various cloud infrastructures and use cases depending on the context. In the process of contextualization, the cloud infrastructure makes the context available to the virtual machine and the virtual machine interprets the context. On contextualization, the virtual machine can, for instance, start certain services, create users, or set configuration parameters. For contextualization, we distinguish between so called meta-data and user-data. The meta-data is provided by the cloud infrastructure and is not modifiable by the user. For example, meta-data includes the instance ID as issued by the cloud infrastructure and the virtual machine's assigned network parameters. The user-data is provided by the user on creation of the virtual machine and an example is shown in Figure 4.

```
[amiconfig]
plugins=cernvm

[cernvm]
repositories=atlas,atlas-condb,atlas-nightlies,grid,sft
shell=/bin/bash
config_url=http://cernvm.cern.ch/config
environment=ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase
keyboard=us-acentos
```

Figure 4: Example of user-data

Meta-data and user-data are typically accessible through an HTTP server on a private IP address such as 169.254.169.254. The cloud infrastructure shields user-data from different VMs so that it can be used to deliver credentials to a virtual machine.

4.1.1 Optimization of Fetching User-data

In the original version, a way to download necessary data is to use **netcat** to create a TCP socket that is used in order to connect to a server. A disadvantage in this approach is checking if it is connected to a specific server whenever downloading is needed. In this case, that is happening four times since there are different options for user data. That is 169.254.169.254

for OpenStack, Amazon EC2, and Google Compute Engine, 10.1.1.1 or the DHCP server for CloudStack.

Idea to optimize this approach is to write a C program that would get a list of servers and timeout to wait for connection as arguments. In this program, sockets would be created, as it is described in [4], that would check connections with all servers in parallel using pthreads. Return value is a bit combination containing '0' or '1' depending on whether it is connected or not. That makes it faster and decrements number of connections checking, in this case, one check does the work that four of them do in the original version.

4.2 Pinning System Files

In order to switch to a real operating system, some system files need to be pinned to the file system cache before. Before any modification, **netcat** was used. This networking utility can be used for creating TCP/UDP connections and investigating them. The biggest use of this utility is in the scripts where we need to deal with TCP/UDP sockets. Its usage has already been shown in the case of contextualization in the previous subsection. In this step of a booting process, Unix Domain Protocol socket described in [5] is created to connect to the file system cache where system files should be pinned. The problem is that while pinning, the original way blocks the whole boot process.

Idea to optimize this approach is to pin the files in parallel before booting ends and let the whole process executes at the same time. The way to achieve this is to create a C program again, but this time daemonized. Running in parallel is succeeded using forking. As it was explained earlier, fork creates a child process that is the same as parent process is and then able to execute program with some modifications. By exiting the parent process, child continues its execution and pins the files in parallel while the boot process is running normally without interruptions. In this daemonized C program, sockets are created for every file that needs to be pinned (154 in μCernVM). When they are connected, messages "pin $file" are written to the socket, i.e. by reading these messages, server side of the socket pins the files.

5 Results

5.1 First Boot - Contextualization

Command	$\sum$ [ms]	#repeats	AVG [ms]
modprobe -q floppy	6063	1	6063
ls ${AUFS_RO}...	2502	1	2502
timeout -t 5 ntpclient -s -h ${NTP_PEER}...	1320	1	1320
nc local:/${ROOT}/mnt/.rw/cache...	1254	154	8
echo -n "pin ${file}"	688	154	4
TOTAL DURATION:			~20.28 s
# DIFFERENT COMMANDS:			155
# COMMANDS:			1094

Table 3: Execution times for a first boot

5.2 Reboot - Contextualization

Command	$\sum$ [ms]	#repeats	AVG [ms]
modprobe -q floppy	6059	1	6059
timeout -t 5 ntpclient -s -h ${NTP_PEER}...	2511	1	2511
nc local:/${ROOT}/mnt/.rw/cache...	1219	154	8
echo -n "pin ${file}"	564	154	4
TOTAL DURATION:			~14.35 s
# DIFFERENT COMMANDS:			128
# COMMANDS:			758

Table 4: Execution times for a reboot

5.3 Reboot - Pinning System Files

Command	$\sum$ [ms]	#repeats	AVG [ms]
modprobe -q floppy	6061	1	6061
timeout -t 5 ntpclient -s -h ${NTP_PEER}...	1001	1	1001
ls ${AUFS_RO}...	585	1	585
mount -o ro /dev/${DEV}...	359	1	359
TOTAL DURATION:			∼11.29 s
# DIFFERENT COMMANDS:			124
# COMMANDS:			254

Table 5: Execution times for a reboot

6 Discussion

By analyzing results obtained after optimization related to contextualization, it is obvious that a first boot takes cca 5 seconds less to execute, as it is expected. In the case of a reboot, this optimized step is skipped since there is no need for downloading user and meta data again, and total execution time shown in Table 4 takes almost the same as in the case shown in Table 2.

Results from a reboot with optimization of pinning files is shown in Table 5 and it can be seen that total execution takes cca 2 seconds less than ones shown in Table 2 and Table 4 because of running processes in parallel, while being daemonized.

7 Conclusion

The aim of this project was to optimize μCernVM early boot process since it has great significance for lots of physicists participating in four large LHC experiments.

Modified BusyBox ash was used as an interpreter of shell scripts that were executing in the early phase. The results of running a machine using this modified tool after optimization of demanding steps, both for a first boot and reboot were given. They showed improvement and successful optimization.

As a future work, a way to get current time can be changed. It is visible from the tables with results that command 'timeout -t 5 ntpclient -s -h

${NTP_PEER}...' takes too much time because it tries to connect to NTP server every time, and connecting to network is time demanding. Idea to optimize this is to set the right time from the hypervisor once at the boot.

When having results for all executed steps and its time consumption, it is easier to find hotspots to optimize, as it was successfully done in this project.

References

[1] Jakob Blomer, Dario Berzano, Predrag Buncic, Ioannis Charalampidis, Gerardo Ganis, Georgios Lestaris, René Meusel, and V Nicolaou. Micro-cernvm: slashing the cost of building and deploying virtual machines. In *Journal of Physics: Conference Series*, volume 513, page 032009. IOP Publishing, 2014.

[2] Jakob Blomer, Predrag Buncic, and René Meusel. The cernvm file system. Technical report, Technical Report, 2013.

[3] Predrag Buncic, Carlos Aguado-Sanchez, Jakob Blomer, and Artem Harutyunyan. Cernvm: Minimal maintenance approach to virtualization. In *Journal of Physics: Conference Series*, volume 331, page 052004. IOP Publishing, 2011.

[4] Michael J Donahoo and Kenneth L Calvert. *The pocket guide to TCP/IP sockets: C version*. Morgan Kaufmann, 2001.

[5] W Richard Stevens, Bill Fenner, and Andrew M Rudoff. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[6] Nicholas Wells. Busybox: A swiss army knife for linux. *Linux Journal*, 2000(78es):10, 2000.

A General Application of Modified BusyBox

Modified BusyBox ash is not only suitable for optimization of CernVM, it can be used as a performance profiler for most of shell scripts, taking into account that it doesn't provide whole set of UNIX utilities. Steps that need to be executed are related to changing bash intepreter to BusyBox interpreter and creating an environment variable as a path to save obtained data to.

replace /bin/bash and /bin/sh with BusyBox:

```
cp <BusyBox BINARY> /bin/BusyBox
mv /bin/bash /bin/bash.orig
ln -s /bin/BusyBox /bin/bash
export ASH_SCORES=/path/to/directory
```