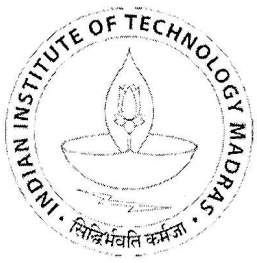
GENERATION OF NEW MOOL KERNEL 3.19.0

A Project Report
Submitted by

M.VENKATESH
G.N.D.VENKATESH
L.SAI SUBRAHMANYAM



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS
MARCH, 2016

1 ACKNOWLEDGEMENTS

We wish to express our sincere gratitude to everyone who contributed in our research work directly or indirectly.

We thank Indian Institute of Technology Madras for giving us an opportunity to pursue our project work here. We are grateful to the Department of Computer Science (IIT Madras), for providing the world class facilities for our research work. This work would not have been possible without the support of our supervisor, **Prof. D. JANAKI RAM.** We wish to thank him for his valuable guidance. His valuable suggestions and directions were indispensable for the completion of our work.

We wish to thank **K BALAJI** and **K MAHENDER** for their help. Their immensely helpful discussions and interactions with them led the path to this thesis. We wish to thank our lab mates for their inputs and technical ideas that helped us a lot.

2.ABSTRACT

KEYWORDS: MOOL, COMPILATION, GIT, MERGE.

Linux kernel is one of the most popular and widely used open source operation systems. It has been developed in a procedural manner with the primary focus on the system performance. As a side effect, there are maintainability issues arising due to high coupling in the kernel. The entire kernel source code can be divided into two parts: the core kernel and the device drivers. Given Linux kernel's popularity, it is deployed on a wide range of machines and supports numerous hardware devices. The device drivers constitute the majority of the kernel code and are also responsible for the majority of the kernel bugs that are found.

MOOL (Minimalistic Object Oriented Linux) aims at redesigning the Linux kernel to reduce coupling and increase maintainability by means of OO (Object Oriented) abstractions. MOOL features a device driver framework to write drivers in C++ and insert them as loadable kernel modules. BOSS-MOOL is intended to support C++ device drivers. This feature is a new offering of BOSS to the Linux community. C++ wrappers have been built for the core of the kernel of Linux operating system and it can be used by kernel module programmers for the developing of their programs in C++, which enhances maintainability and reuse for their programs.

The three tasks that we have done are presented in the paper

TABLE OF CONTENTS

3. KERNEL COMPILATION

In this chapter we are going to explain about the compilation process that is to be done.

3.1 DEVELOPING A (.config) FILE:

The .config file is developed after defining which drivers are to enabled as modules during the process of compilation. The configuration file is used to configure the parameters and initialise the settings of device drivers for the kernel. To generate a .config file we have to follow the following commands.We can generate a .config file using several make <option>config commands.The most commonly used commands are:

make menuconfig -

This allows the user to configure in the text editor mode with low GUI. After giving this command a dialog box will be displayed which shows us to enable or disable the drivers. We can enable the drivers as two types, one by giving it as an inbuilt thing and the second is by giving it as a module thing. To enable it as a module we have to give **[M]**, to enable it as an inbuilt feature we have to give **[*]**, if we leave it blank **[ ]** then it represents the driver is not initialised.

make xconfig -

This allows the user to configure by selecting the boxes which is provided with more GUI. By selecting the required box it enables the user to enable the feature.

make defconfig -

This allows the user to generate a configuration file with default configuration.

3.2 COMPILING THE KEREL USING "make":

make is a build animation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Make can also be used to manage any project where some files must be updated automatically from others whenever the others change. After running the command **make menuconfig** and generating a **.config** file we have to run the command **make** so that the **'.c'** files are compiled to **'.o'** to produce a build executable program.

3.3 INSTALLING THE MODULES :

After running the **make** command we have to run the command **sudo make modules_install** so that the modules of necessary drivers that are enabled are installed. After this we have to give the command **sudo make install** which generates a kernel version that is compiled. **make install** invokes make, make finds the target install in Makefile and files the directions to install the program.

3.4 GENERATING THE INITRDIMG:

To generate the initrdimg we have to give the following command **sudo mkinitramfs -o /boot/initrdimg- (kernel version). mkinitramfs** is a low-level tool for generating an initramfs image, it is used to mount the root file system. **initrdimg** is an initial root file system that is the mounted prior to when the real file system is available. **initrd** is bound to kernel and loaded as a part of kernel boot procedure. It produces the capability to lead the RAM disk by the boot loader.

3.5 UPDATING THE GRUB AND RUNNING THE KERNEL:

After generating the initrdimg we have to update the grub by using the command **sudo update-grub** so that the generated new version is loaded at the process of starting in the grub boot loader. GNU GRUB (GNU GRand Unified Bootloader) is a boot loader is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular operating system's partitions. We can view the booted kernel in the terminal by using the command **"uname -a".**

4. USING GIT AND MAINTAINING REPOSITORY

4.1 Initialising a repository:

Command: git init

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the .git directory, an existing project remains unaltered (unlike SVN, Git doesn't require a .git folder in every subdirectory).

Transform the current directory into a Git repository. This adds a .git folder to the current directory and makes it possible to start recording revisions of the project

4.2 Cloning the repository

Command: git clone

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the working copy is a full-fledged Git repository. It has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

4.3 Viewing and selecting the branches in the repository

Command : git branch

"git branch" should show all the local branches of your repo. The starred branch is your current branch.

Command: git checkout

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out

branches. In this module, were only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

Usage

git checkout master

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the current state of the project.

git checkout <commit ><file >

Check out a previous version of a file. This turns the <file >that resides in the working directory into an exact copy of the one from <commit >and adds it to the staging area.

git checkout <commit >

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit >argument. This will put you in a detached HEAD start.

4.4 Git update & merge:

To update your local repository to the newest commit, execute

$ git pull

In your working directory to *fetch* and *merge* remote changes. to merge another branch into your active branch (e.g. master), use

$ git merge <branch>

In both cases git tries to auto-merge changes. Unfortunately, this is not always possible and results in *conflicts*. You are responsible to merge those *conflicts* manually by editing the files shown by git. After changing, you need to mark them as merged with

$ git add <filename>

Before merging changes, you can also preview them by using **$ git diff <source branch><target branch>.**

4.5 Add & commiting changes

You can propose changes (add it to the **Index**) using
$ git add <file name>
$ git add *
This is the first step in the basic git workflow. To actually commit these changes use
$git commit -m "commit message"
Now the file is committed to the **HEAD**, but not in your remote repository yet.

f.pushing changes:

4.6. Pushing changes

Your changes are now in the **HEAD** of your local working copy. To send those changes to your remote repository, execute
$ git push origin master
Change *master* to whatever branch you want to push your changes to.
If you have not cloned an existing repository and want to connect your repository to a remote server, you need to add it with
$ Git remote add origin <server>
Now you are able to push your changes to the selected remote server

5. EXTENDING THE MOOL SUPPORT AND BUILDING OF THE MOOL KERNEL 3.19.0

5.1. Cloning the MOOL repo:

Initially the MOOL repository containing the 3.18.3 kernel which is having the MOOLsupport is to be taken. It is done by cloning it from the source repository using the command - " git clone "followed by the git address. Now change to the directory which is cloned by using the command - " cd directory_name ".

5.2. Compile and build the kernel:

Compile the 3.18.3 kernel which is cloned in the following way. Initially run the command - " make menuconfig ". This command allows the user to enable the desired configuration.This generates a .config file. After generating the .config file give the command - " make ". It is used as a build animation tool that automatically builds executable programs and the libraries from the source code by reading Makefiles which specify how to derive the target program.

5.3. Install the modules:

After the compilation give the command - " sudo make modules_install " to install the necessary modules.

Next to this command give the command - "sudo make install " which shows the compiled kernel version.

5.4 Generation of initrdimg and running the kernel:

Now generate the kernel version image by using the command - " sudo mkinitramfs -o /boot/initrdimg-(generated kernel version) ".

Update the grub inorder to add the generated kernel version to the grub by using the command - " sudo update-grub ". Reboot the system and start the new kernel version developed.

5.5 Extending the MOOL support to other kernel version:

To extend the MOOL support to the new kernel version first we have to take a new branch from the repository by using the command - " git checkout branch_name ". If the branch is not present there we have to create it by using the command

" $git checkout -b branch_name".

To check the current status of the working branch use the command - " git status ". To check the branch present in the repo we have to give the command

-$ " git tag -l ".

Then merge the 3.18.3 kernel which is having the MOOL support with the branch 3.18.4 . Use $ git merge to merge the kernels.

Later then compile the 3.18.4 branch just as we have done for 3.18.3 and run it .

Repeat the same process till 3.18.9 as there are no conflicts araised during the merging and compilation part. (i.e v 3.18.4/5/6/7/8/9)

5.6 Conflicts of merging and compiling version 3.19.0:

While merging:

Now while merging the 3.18.9 branch with 3.19.0 we get conflicts - 1.new files,2.modified files, 3.bothmodified files. New code is appended to the files present in the head branch which leads to conflicts while giving the make command.

Solution:

The new files are added to the branch by using the command - " git add -A " or " git add (filename) ". Then commit the changes to the branch by using the command - " git commit -m " first commit" ".

While compiing:

Then give the command - " make ". The following are the conflicts araised during the make process.

Fatal error :Missing separator , ;

Error: not recognized as a symbol >;

Solution:

These conflicts are solved by removing the parenthesis " <<<<<< head " and the text between

" ======= and 3.18.9>>>>>>>" in the conflict files as they are appended while merging by considering the MOOL changes and by including the header files with reference to the Linux reference code in the server. Here head represents the current working branch i.e, 3.19.0 and the lines of code below it represents the 3.19.0 branch so it should not be removed where

as the code between " 3.18.9 " represents the 3.18.9 branch which can be removed as it is already there.

After resolving these conflicts repeat the same procedure by giving the " make " command and generate the new kernel i.e, 3.19.0. you can check the booted MOOL kernel in the terminal by using the command "uname -a".

6.CONCLUSION

Linux kernel is one of the most popular and widely used open source operation systems. It has been developed in a procedural manner with the primary focus on the system performance. As a side effect, there are maintainability issues arising due to high coupling in the kernel. The entire kernel source code can be divided into two parts : the core kernel and the device drivers. Given Linux kernel popularity, it is deployed on a wide range of machines and supports numerous hardware devices. The device drivers constitute the majority of the kernel code and are also responsible for the majority of the kernel bugs that are found. MOOL Kernel has been developed at DOS (Distributed and Object Systems) Lab, IIT, Madras. MOOL features a device driver framework to write drivers in C++ and insert them as loadable kernel modules.

DOS Lab at IIT Madras and CDAC Chennai have integrated the MOOL kernel into BOSS MOOL distribution. BOSS MOOL supports C++ device drivers. This feature is a new offering of BOSS to the Linux community.Further, the BOSS-MOOL Kernel supports localisation at Console level, which hitherto was not available on any flavors of LINUX.The students can also develop Message filters. Message filters work by intercepting system calls, built using object-oriented wrappers which provide better maintainability and extensibility, without modifying the existing kernel code. Placing them at the system call level disturbs very little of the kernel code.Filters are developed as kernel modules, so they can be dynamically added at runtime. When a user makes a system call, it is forwarded transparently to the filter which in-turn will call the actual system call. The users and applications are not aware of the presence of filter model, as there is no change to the user interface. Hence this design provides a transparent way of intercepting messages for the kernel.