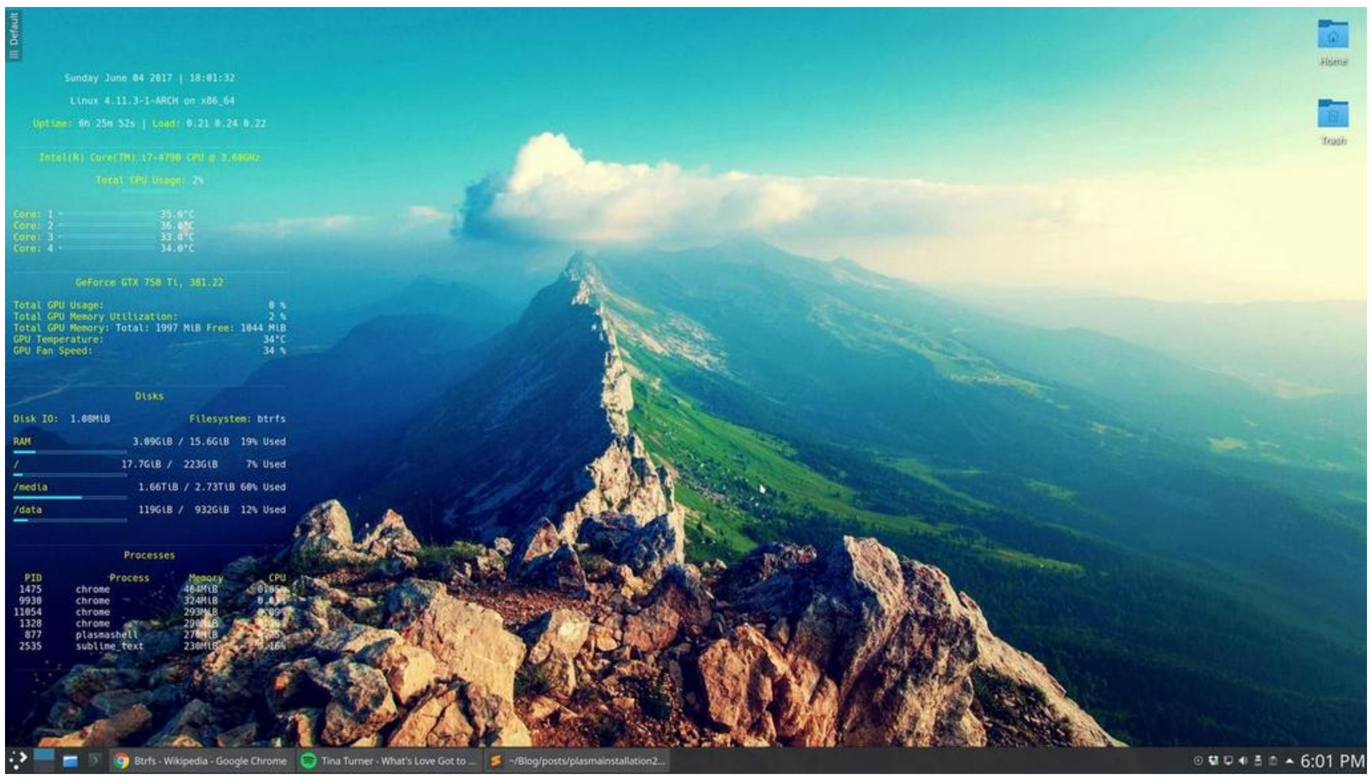
# My Arch Linux Setup with Plasma 5

By Sadanand Singh — on 2017-06-05 in Computers — Comments

Arch Linux is a general purpose GNU/Linux distribution that provides most up-to-date software by following the rolling-release model. Arch Linux allows you to use updated cutting-edge software and packages as soon as the developers released them. KDE Plasma 5 is the current generation of the desktop environment created by KDE primarily for Linux systems.

In this post, we will do a complete installation of Arch Linux with Plasma 5 as the desktop environment. Our setup will also involve encryption of the root partition that will be formatted in btrfs. This post is an updated and a more complete version of my previous posts on Arch Linux and Plasma 5 Installation.

**Table of Contents**



```
Sunday June 04 2017 | 18:01:32
Linux 4.11.3-1-ARCH on x86_64
Uptime: 0h 25m 52s | Load: 0.21 0.24 0.22

Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
Total CPU Usage: 2%

Core: 1              35.0°C
Core: 2              36.0°C
Core: 3              33.0°C
Core: 4              34.0°C

GeForce GTX 750 Ti, 381.22

Total GPU Usage:                              0 %
Total GPU Memory Utilization:                 2 %
Total GPU Memory: Total: 1997 MiB Free: 1044 MiB
GPU Temperature:                             34°C
GPU Fan Speed:                               34 %

Disks

Disk IO:  1.08MiB               Filesystem: btrfs

RAM                  3.09GiB / 15.6GiB  19% Used
/                   17.7GiB /  223GiB    7% Used
/media               1.66TiB / 2.73TiB 60% Used
/data                119GiB /  932GiB  12% Used

Processes

  PID      Process            Memory      CPU
 1475     chrome             464MiB     [illegible]
 9930     chrome             324MiB     [illegible]
11054     chrome             293MiB     [illegible]
 1328     chrome             290MiB     [illegible]
  877     plasmashell        270MiB     [illegible]
 2535     sublime_text       230MiB     [illegible]
```

Default
Home
Trash
Btrfs - Wikipedia - Google Chrome
Tina Turner - What's Love Got to ...
~/Blog/posts/plasmainstallation2...
6:01 PM

## System Details

For reference, my installation system is a slightly upgraded form of my original desktop:

- i7 4790 3.6 GHz (Haswell)
- ASRock Z97 Extreme6 LGA 1150 Intel Z97 HDMI SATA USB 3.0
- ADATA XPG V1.0 DDR3 1866 4x4 GB RAM
- OCZ Vertex 460A Series 2.5” 240 GB
- WD Blue 1TB 3.5” 7200 RPM, 64MB Cache
- WD Blue 3TB 3.5” 7200 RPM, 64MB Cache
- Ultra LSP V2 650 Watt PSU
- Cooler Master - MasterCase Pro 5
- Asus BW-12B1ST/BLK/G/AS Blue Ray Burner
- Samsung U28E590D 28-Inch UHD LED-Lit 4K Monitor
- Nvidia GeForce GTX 750 Ti GPU

## Base Installation

**NOTE**

I do not wish to repeat Arch Installation Guide here.

Do not forget about Arch Wiki, the best documentation in the world! Most of the content in this post has been compiled from the Arch wiki.

Before beginning this guide, I would assume that you have a bootable USB of the latest Arch Linux Installer. If not, please follow the Arch wiki guide to create one for you.

Once you login in the installer disk, You will be logged in on the first virtual console as the root user, and presented with a *zsh* shell prompt. I will assume you have an Ethernet connection and hence will be connected to Internet by default. If you have to rely on wifi, please refer to the Wireless Network Configuration wiki page for the detailed setup. **You must have Internet connection at this stage before proceeding any further.**

You should boot into *UEFI* mode if you have a UEFI motherboard and UEFI mode enabled.

To verify you have booted in UEFI mode, run:

```
$ efivar -l
```

This should give you a list of set UEFI variables. Please look at the Arch Installation Guide in case you do not get any list of UEFI variables.

The very first thing that annoys me in the virtual console is how tiny all the fonts are. We will fix that by running the following commands:

```
$ pacman -Sy
$ pacman -S terminus-font
$ setfont ter-132n
```

We are all set to get started with the actual installation process.

## HDDs Partitioning

First find the hard drive that you will be using as the main/root disk.

```
$ cat /proc/partitions

# OUTPUT eg.
# major minor  #blocks  name

# 8        0  268435456 sda
# 9        0  268435456 sdb
# 19       0  268435456 sdc
# 11       0     759808 sr0
# 7        0     328616 loop0
```

Say, we will be using */dev/sda* as the main disk and */dev/sdb* as */data* and */dev/sdc* as */media* .

Because we are creating an encrypted file system it’s a good idea to overwrite it with random data.

We’ll use **badblocks** for this. Another method is to use *dd if=/dev/urandom of=/dev/xxx*, the *dd* method is probably the best method, but is a lot slower. **The following step should take about 20 minutes on a 240 GB SSD.**

```
$ badblocks -c 10240 -s -w -t random -v /dev/sda
```

Next, we will create GPT partitions on all disks using *gdisk* command.

```
$ dd if=/dev/zero of=/dev/sda bs=1M count=5000
$ gdisk /dev/sda
Found invalid MBR and corrupt GPT. What do you want to do? (Using the
GPT MAY permit recovery of GPT data.)
 1 - Use current GPT
 2 - Create blank GPT
```

Then press 2 to create a blank GPT and start fresh

```
ZAP:
$ press x - to go to extended menu
$ press z - to zap
$ press Y - to confirm
$ press Y - to delete MBR
```

It might now kick us out of *gdisk*, so get back into it:

```
$ gdisk /dev/sda

$ Command (? for help): m
$ Command (? for help): n

$ Partition number (1-128, default 1):
$ First sector (34-500118158, default = 2048) or {+-}size{KMGTP}:
$ Last sector (2048-500118, default = 500118) or {+-}size{KMGTP}: 512M
$ Current type is 'Linux filesystem'
$ Hex code or GUID (L to show codes, Enter = 8300): ef00
$ Changed type of partition to 'EFI System'

$ Partition number (2-128, default 2):
$ First sector (34-500118, default = 16779264) or {+-}size{KMGTP}:
$ Last sector (16779264-500118, default = 500118) or {+-}size{KMGTP}:
$ Current type is 'Linux filesystem'
$ Hex code or GUID (L to show codes, Enter = 8300):
$ Changed type of partition to 'Linux filesystem'

$ Command (? for help): p
$ Press w to write to disk
$ Press Y to confirm
```

Repeat the above procedure for */dev/sdb* and */dev/sdc*, but create just one partition with all values as default. At the end we will have three partitions: */dev/sda1*, */dev/sda2*, */dev/sdb1* and */dev/sdc1*.

## Setup Disk Encryption

Our /boot partition will be on */dev/sda1*, while the main installation will be on */dev/sda2*. In this setup, we will be enabling full encryption on */dev/sda2* only.

In order to enable disk encryption, we will first create a root luks volume, open it and then format it.

```
# first, we need to prepare the encrypted (outer) volume
$ cryptsetup --cipher aes-xts-plain64 --hash sha512 --use-random --verify-passphrase luksFormat /dev/sda2

# I really hope I don't have to lecture you on NOT LOSING this
# password, lest all of your data will be forever inaccessible,
# right?

# then, we actually open it as a block device, and format the
# inner volume later
$ cryptsetup luksOpen /dev/sda2 root
```

**Automatic Key Login from an USB/SD Card**

If you want to automatically login the encrypted disk password from an externally attached USB or SD card, you will first need to create a key file.

```
$ dd bs=512 count=4 if=/dev/urandom of=KEYFILE
```

Then, add this key to the luks container, so that it can be later used to open the encrypted drive.

```
$ cryptsetup luksAddKey /dev/sda2 KEYFILE
```

Note that the KEYFILE here should be kept on a separate USB drive or SD card.

The recommended way of using such a disk would be as follows:

```
# assuming our USB of interest is /dev/sdd  and can be format
#
# Format the drive
$ dd if=/dev/zero of=/dev/sdd bs=1M
# Create partitions using gdisk
#
$ gdisk /dev/sdd
#
# Follow along to create one partition (/dev/sdd1) of type 0700
#
# format /dev/sdd1
$ mkfs.fat /dev/sdd1

# mount the newly format disk on /mnt and then copy the KEYFILE
$ mount /dev/sdd1 /mnt
$ mv KEYFILE /mnt/KEYFILE
$ umount /mnt
```

We will be later using this KEYFILE in boot loader setup.

## Format HDDs

At this point, we have following drives ready for format: */dev/sda1*, */dev/mapper/root*, */dev/sdb1* and */dev/sdc1*.

These can be format as follows:

```
$ mkfs.vfat -F32 /dev/sda1
$ mkfs.btrfs -L arch /dev/mapper/root
$ mkfs.btrfs -L data /dev/sdb1
$ mkfs.btrfs -L media /dev/sdc1
```

Now, we will create *btrfs* subvolumes and mount them properly for installation and final setup.

```
$ mount /dev/mapper/root /mnt
$ btrfs subvolume create /mnt/ROOT
$ btrfs subvolume create /mnt/home
$ umount /mnt

$ mount /dev/sdb1 /mnt
$ btrfs subvolume create /mnt/data
$ umount /mnt

$ mount /dev/sdc1 /mnt
$ btrfs subvolume create /mnt/media
$ umount /mnt
```

Now, once the sub-volumes have been created, we will mount them in appropriate locations with optimal flags.

```
$ SSD_MOUNTS="rw,noatime,nodev,compress=lzo,ssd,discard,
space_cache,autodefrag,inode_cache"
$ HDD_MOUNTS="rw,nosuid,nodev,relatime,space_cache"
$ EFI_MOUNTS="rw,noatime,discard,nodev,nosuid,noexec"
$ mount -o $SSD_MOUNTS,subvol=ROOT /dev/mapper/root /mnt
$ mkdir -p /mnt/home
$ mkdir -p /mnt/data
$ mkdir -p /mnt/media
$ mount -o $SSD_MOUNTS,nosuid,subvol=home /dev/mapper/root /mnt/home
$ mount -o $HDD_MOUNTS,subvol=data /dev/sdb1 /mnt/data
$ mount -o $HDD_MOUNTS,subvol=media /dev/sdc1 /mnt/media

$ mkdir -p /mnt/boot
$ mount -o $EFI_MOUNTS /dev/sda1 /mnt/boot
```

Save the current */etc/resolv.conf* file for future use!

```
$ cp /etc/resolv.conf /mnt/etc/resolv.conf
```

## Base System Installation

Now, we will do the actually installation of base packages.

```
$ pacstrap /mnt base base-devel btrfs-progs
$ genfstab -U -p /mnt >> /mnt/etc/fstab
```

## Initial System Setup

Edit the */mnt/ect/fstab* file to add following */tmp* mounts.

```
tmpfs /tmp tmpfs rw,nodev,nosuid 0 0
tmpfs /dev/shm tmpfs rw,nodev,nosuid,noexec 0 0
```

Finally bind root for installation.

```
$ arch-chroot /mnt "bash"
$ pacman -Syy
$ pacman -Syu
$ pacman -S sudo vim
$ vim /etc/locale.gen

...
# en_SG ISO-8859-1
en_US.UTF-8 UTF-8
# en_US ISO-8859-1
...

$ locale-gen
$ echo LANG=en_US.UTF-8 > /etc/locale.conf
$ export LANG=en_US.UTF-8
$ ls -l /usr/share/zoneinfo
$ ln -sf /usr/share/zoneinfo/Zone/SubZone /etc/localtime
$ hwclock --systohc --utc
$ sed -i "s/# %wheel ALL=(ALL) ALL/%wheel ALL=(ALL) ALL/" /etc/sudoers
$ HOSTNAME=euler
$ echo $HOSTNAME > /etc/hostname
$ passwd
```

We will also add *hostname* to our `/etc/hosts` file:

```
$ vim /etc/hosts
...
127.0.0.1       localhost.localdomain   localhost
::1             localhost.localdomain   localhost
127.0.0.1       $HOSTNAME.localdomain   $HOSTNAME
...
```

We also need to fix the `mkinitcpio.conf` to contain what we actually need.

```
$ vi /etc/mkinitcpio.conf
# on the MODULES section, add "vfat aes_x86_64 crc32c-intel"
# (and whatever else you know your hardware needs. Mine needs i915 too)
# on the BINARIES section, add "/usr/bin/btrfsck", since it's useful
# to have in case your filesystem has troubles
# on the HOOKS section:
#  - add "encrypt" before "filesystems"
#  - remove "fsck" and
#  - add "btrfs" at the end
#
# re-generate your initrd images
mkinitcpio -p linux
```

## Boot Manager Setup

*systemd-boot*, previously called *gummiboot*, is a simple UEFI boot manager which executes configured EFI images. The default entry is selected by a configured pattern (glob) or an on-screen menu. It is included with the *systemd*, which is installed on an Arch systems by default.

Assuming */boot* is your boot drive, first run the following command to get started:

```
$ bootctl --path=/boot install
```

It will copy the systemd-boot binary to your EFI System Partition ( `/boot/EFI/systemd/systemd-bootx64.efi` and `/boot/EFI/Boot/BOOTX64.EFI` - both of which are identical - on **x64** systems ) and add *systemd-boot* itself as the default EFI application (default boot entry) loaded by the EFI Boot Manager.

Finally to configure out boot loader, we will need the UUID of some of our hard drives. These can be easily done using the *blkid* command.

```
$ blkid /dev/sda1 > /boot/loader/entries/arch.conf
$ blkid /dev/sda2 >> /boot/loader/entries/arch.conf
$ blkid /dev/mapper/root >> /boot/loader/entries/arch.conf
$ blkid /dev/sdd1 >> /boot/loader/entries/arch.conf

# for this example, I'm going to mark them like this:
# /dev/sda1 LABEL="EFI"              UUID=11111111-1111-1111-1111-111111111111
# /dev/sda2 LABEL="arch"     UUID=33333333-3333-3333-3333-333333333333
# /dev/mapper/root LABEL="Arch Linux"  UUID=44444444-4444-4444-4444-444444444444
# /dev/sdd1 LABEL="USB"     UUID=0000-0000  # this is the drive where KEYFILE exists
```

Now, make sure that the following two files look as follows, where UUIDs is the value obtained from above commands.

Do not forget to modify UUIDs and KEYFIL entries!

```
$ vim /boot/loader/loader.conf
...
timeout 3
default arch
...
$ vim /boot/loader/entries/arch.conf
...

title Arch Linux
linux /vmlinuz-linux
initrd /initramfs-linux.img
options ro cryptdevice=UUID=33333333-3333-3333-3333-333333333333:luks-33333333-3333-3333-3333-333333333333
root=UUID=44444444-4444-4444-4444-444444444444 rootfstype=btrfs rootflags=subvol=ROOT cryptkey=UUID=0000-0000:vfat:KEYFILE
...
```

## Network Setup

At first we will need to figure out the Ethernet controller on which cable is connected.

```
$ networkctl
#
# IDX LINK             TYPE               OPERATIONAL SETUP
#   1 lo               loopback           carrier     unmanaged
#   2 enp3s0           ether              no-carrier  unmanaged
#   3 wlp6s0           wlan               no-carrier  unmanaged
#   4 enp0s25          ether              routable    configured
#
```

In my case, the name of the device is *enp0s25*.

Using this name of the device, we need to configure, and enable the *systemd-networkd.service* service.

Note that we will using the *resolv.conf* that we saved from this session.

Network configurations are stored as *.network in `/etc/systemd/network`. We need to create ours as follows.:

```
$ vim /etc/systemd/network/50-wired.network
$
...
[Match]
Name=enp0s25

[Network]
DHCP=ipv4


...

$
```

Now enable the `networkd` services:

```
systemctl enable systemd-networkd.service
```

Your network should be ready for the first use!

Sync time automatically using the *systemd* service:

```
$ vim /etc/systemd/timesyncd.conf
$
...
[Time]
NTP=0.arch.pool.ntp.org 1.arch.pool.ntp.org 2.arch.pool.ntp.org 3.arch.pool.ntp.org
FallbackNTP=0.pool.ntp.org 1.pool.ntp.org 0.fr.pool.ntp.org
...
$
$ timedatectl set-ntp true
$ timedatectl status
$
...
      Local time: Tue 2016-09-20 16:40:44 PDT
  Universal time: Tue 2016-09-20 23:40:44 UTC
        RTC time: Tue 2016-09-20 23:40:44
       Time zone: US/Pacific (PDT, -0700)
 Network time on: yes
NTP synchronized: yes
 RTC in local TZ: no
 ...
$
```

Avahi is a tool that allows programs to publish and discover services and hosts running on a local network with no specific configuration. For example you can plug into a network and instantly find printers to print to, files to look at and people to talk to.

We can easily set it up it as follows:

```
$ pacman -S avahi nss-mdns
$ systemctl enable avahi-daemon.service
```

We will also install `terminus-font` on our system to work with proper fonts on first boot.

```
$ pacman -S terminus-font
```

## First Boot Installations

Now we are ready for the first boot! Run the following command:

```
$ exit
$ umount -R /mnt
$ reboot
```

After your new system boots, Network should be setup at the start. Check the status of network using:

```
# Set readable font first!
setfont ter-132n
ping google.com -c 2

#
# PING google.com (10.38.24.84) 56(84) bytes of data.
# 64 bytes from google.com (10.38.24.84): icmp_seq=1 ttl=64 time=0.022 ms
# 64 bytes from google.com (10.38.24.84): icmp_seq=2 ttl=64 time=0.023 ms
#
# --- google.com ping statistics ---
# 2 packets transmitted, 2 received, 0% packet loss, time 999ms
# rtt min/avg/max/mdev = 0.022/0.022/0.023/0.004 ms
#
```

If you do not get this output, please follow the troubleshooting links at Arch Wiki on setting up network.

## Adding New User

Choose `$USERNAME` per your liking. I chose *ssingh*, so in future commands whenever you see *ssingh* please replace it with your `$USERNAME`.

```
$ pacman -S zsh
$ useradd -m -G wheel -s usr/bin/zsh $USERNAME
$ chfn --full-name "$FULL_NAME" $USERNAME
$ passwd $USERNAME
```

## GUI Installation with nvidia

I will be assuming you have an `NVIDIA` card for graphics installation.

To setup a graphical desktop, first we need to install some basic X related packages, and some *essential* packages (including fonts):

```
$ pacman -S xorg-server nvidia nvidia-libgl nvidia-settings mesa
```

To avoid the possibility of forgetting to update your *initramfs* after an *nvidia* upgrade, you have to use a *pacman* hook like this:

```
$ vim /etc/pacman.d/hooks/nvidia.hook
$
...
[Trigger]
Operation=Install
Operation=Upgrade
Operation=Remove
Type=Package
Target=nvidia

[Action]
Depends=mkinitcpio
When=PostTransaction
Exec=/usr/bin/mkinitcpio -p linux
...
$
```

Nvidia has a daemon that is to be run at boot. To start the *persistence* daemon at boot, enable the `nvidia-persistenced.service`.

```
$ systemctl enable nvidia-persistenced.service
$ systemctl start nvidia-persistenced.service
```

### How to Avoid Screen Tearing

Tearing can be avoided by forcing a full composition pipeline, regardless of the compositor you are using.

In order to make this change permanent, We will need to edit nvidia configuration file. Since, by default there aren't any, we will first need to create one.

```
$ nvidia-xconfig
$ mv /etc/X11/xorg.cong /etc/X11/xorg.conf.d/20-nvidia.conf
#
# Edit this file as follows:
vim /etc/X11/xorg.conf.d/20-nvidia.conf
# -------------------------------------------
# Section "Screen"
#     Identifier     "Screen0"
#     Option         "metamodes" "nvidia-auto-select +0+0 { ForceFullCompositionPipeline = On }"
#     Option         "AllowIndirectGLXProtocol" "off"
#     Option         "TripleBuffer" "on"
# EndSection
[...]
# Section "Device"
#     [...]
#     Option         "TripleBuffer" "True"
#     [...]
# EndSection
# [...]
# ------------------------------------------------
```

Specific for Plasma 5, we will also create the following file to avoid any tearing in Plasma.

```
$ vim /etc/profile.d/kwin.sh
$
...
export KWIN_TRIPLE_BUFFER=1
...
```

### How to Enable Better Resolution During Boot

The kernel compiled in *efifb* module supports high-resolution nvidia console on EFI systems. This can enabled by enabling the DRM kernel mode setting.

First, we will need to add following to MODULES section of the *mkinitcpio.conf* file:

- *nvidia*
- *nvidia_modeset*
- *nvidia_uvm*
- *nvidia_drm*

We will also need to pass the *nvidia-drm.modeset=1* kernel parameter during the boot.

```
$ vim /etc/mkinitcpio.conf
$
...
MODULES="vfat aes_x86_64 crc32c-intel nvidia nvidia_modeset nvidia_uvm nvidia_drm"
...
$
$ vim /boot/loader/entries/arch.conf
$
...
options ro cryptdevice=UUID=:luks- root=UUID= rootfstype=btrfs rootflags=subvol=ROOT
cryptkey=UUID=:vfat:deepmind20170602 nvidia-drm.modeset=1
...
$
$ mkinitcpio -p linux
```

## Plasma 5 Installation and Setup

We can now proceed with the installation of Plasma 5. In the process, we will also install some useful fonts.

```
$ pacman -S ttf-hack ttf-anonymous-pro
$ pacman -S ttf-dejavu ttf-freefont ttf-liberation
$ pacman -S plasma-meta dolphin kdialog kfind
$ pacman -S konsole gwenview okular spectacle kio-extras
$ pacman -S kompare dolphin-plugins kwallet kwalletmanager
$ pacman -S ark yakuake flite
```

We will also need to select proper themes for the Plasma 5 display manager *sddm* and then enable its *systemd* service.

```
$ vim /etc/sddm.conf

....
[Theme]
# Current theme name
Current=breeze

# Cursor theme used in the greeter
CursorTheme=breeze_cursors
...

$ systemctl enable sddm
$ reboot
```

Once, we boot into the new system, we should have a basic Plasma 5 desktop waiting for you. In the following section, we will be do installation and modifications to the system that I prefer.

## Post Installation Setup

Plasma 5 provides a handy network manager applet. However, in order to use it properly we will need the NetworkManager service to be enabled. This applet allows user specific enabling of *wifi*, *ethernet* or even *VPN* connections.

```
$ sudo pacman -S networkmanager
$ systemctl enable NetworkManager.service
$ systemctl start NetworkManager.service
```

We can also automate the *hostname* setup using the following *systemd* command:

```
$ hostnamectl set-hostname $HOSTNAME
```

## Selecting pacman Mirrors

The *pacman* package provides a “bash” script, */usr/bin/rankmirrors*, which can be used to rank the mirrors according to their connection and opening speeds to take advantage of using the fastest local mirror.

We will do this only on the US based mirrors. First make a copy of the mirrors list file and then delete all non-US mirrors. We will then *rankmirrors* script on the modified list to get the top 6 mirrors for our regular use.

```
$ cp /etc/pacman.d/mirrorlist /etc/pacman.d/mirrorlist.backup
$ cp /etc/pacman.d/mirrorlist /etc/pacman.d/mirrorlist.us
$ vim /etc/pacman.d/mirrorlist.us
....
# Delete all non-US servers
....
$ rankmirrors -n 6 /etc/pacman.d/mirrorlist.us > /etc/pacman.d/mirrorlist
```

## Setup AUR

AUR is a community-driven repository for Arch users. This allows you to install many popular packages that are otherwise not available through core repositories.

In order to make all types of installations uniform, I use pacaur as the preferred tool for installing all packages. One the biggest advantages of *pacaur* is that is uses exactly the same options that regular *pacman* uses.

In order to install *pacuar*, first install dependencies.

```
$ sudo pacman -S expac yajl curl gnupg --noconfirm
```

Create a temp directory for building packages:

```
$ mkdir ~/temp
$ cp ~ temp
```

Install *cower* first and then *pacaur*:

```
$ gpg --recv-keys --keyserver hkp://pgp.mit.edu 1EB2638FF56C0C53
$ curl -o PKGBUILD https://aur.archlinux.org/cgit/aur.git/plain/PKGBUILD?h=cower
$ makepkg -i PKGBUILD --noconfirm

$ curl -o PKGBUILD https://aur.archlinux.org/cgit/aur.git/plain/PKGBUILD?h=pacaur
$ makepkg -i PKGBUILD --noconfirm

# Finally cleanup and remove the temp directory
$ cd ~
$ rm -r ~/temp
```

## Audio Setup

This is pretty simple. Install following packages and you should be done:

```
$ sudo pacaur -S alsa-utils pulseaudio pulseaudio-alsa mpv
$ sudo pacaur -S libcanberra-pulse libcanberra-gstreamer
$ sudo pacaur -S vlc-qt5
```

Now start the *pulseaudio* service.

```
$ systemctl --user enable pulseaudio.socket
```

## Web Browsers

My preferred choice of browsers is *google chrome*. However, it is also good to have the KDE native *qupzilla*.

```
$ sudo pacaur -S google-chrome qupzilla
```

*Profile-sync-daemon (psd)* is a tiny pseudo-daemon designed to manage browser profile(s) in *tmpfs* and to periodically sync back to the physical disc (HDD/SSD). This is accomplished by an innovative use of *rsync* to maintain synchronization between a *tmpfs* copy and media-bound backup of the browser profile(s). These features of *psd* leads to following benefits:

- Transparent user experience
- Reduced wear to physical drives, and
- Speed

To setup. first install the *profile-sync-daemon* package.

```
sudo pacaur -S profile-sync-daemon
```

Run *psd* the first time which will create a configuration file at \$XDG_CONFIG_HOME/psd/psd.conf which contains all settings.

```
$ psd
# First time running psd so please edit
# /home/$USERNAME/.config/psd/psd.conf to your liking and run again.
```

In the config file change the BROWSERS variables to *google-chrome qupzilla*. Also, enable the use of *overlayfs* to improve sync speed and to use a smaller memory footprint. Do this in the *USE_OVERLAYFS="yes"* variable.

Note: USE_OVERLAYFS feature requires a Linux kernel version of 3.18.0 or greater to work.

In order to use the OVERLAYFS feature, you will also need to give *sudo* permissions to psd-helper as follows (replace `$USERNAME` accordingly):

```
$ vim /etc/sudoers
...
$USERNAME ALL=(ALL) NOPASSWD: /usr/bin/psd-overlay-helper
...
```

Verify the working of configuration using the preview mode of psd:

```
psd p
```

*Google Chrome* by default uses *kdewallet* to manage passwords, where as *Qupzilla* does not. You can change that in its settings.

## git Setup

Install git and setup some global options as below:

```
$ sudo pacaur -S git
$
$ vim ~/.gitconfig
...
[user]
    name = Sadanand Singh
    email = EMAIL_ADDRESS
[color]
    ui = auto
[status]
    showuntrackedfiles = no
[alias]
    gist = log --graph --oneline --all --decorate --date-order
    find = log --graph --oneline --all --decorate --date-order --regexp-ignore-case --extended-regexp --grep
    rfind = log --graph --oneline --all --decorate --date-order --regexp-ignore-case --extended-regexp --invert-grep --grep
    search = grep --line-number --ignore-case -E -I
[pager]
    status = true
[push]
    default = matching
[merge]
    tool = meld
[diff]
    tool = meld

[help]
    autocorrect = 1
...
```

## ssh Setup

To get started first install the *openssh* package.

```
sudo pacaur -S openssh
```

The ssh server can be started using the *systemd* service. Before starting the service, however, we want to generate ssh keys and setup the server for login based only on keys.

```
$ ssh-keygen -t ed25519
$
# Create a .ssh/config file for rmate usage in sublime text
$ vim ~/.ssh/config
...
RemoteForward 52698 localhost:52698
...
$
# Create ~/.ssh/authorized_keys file with list of machines that
# are allowed to login to this machine.
$ touch ~/.ssh/authorized_keys
$
# Finally edit the /etc/ssh/sshd_config
# file to disable Password based logins
$ sudo vim /etc/ssh/sshd_config
...
PasswordAuthentication no
...
```

Furthermore, before enabling the *sshd* service, please also ensure to copy your keys to all your relevant other servers and places like github.

We can now use *systemd* to start the ssh service.

```
$ systemctl enable sshd.socket
$ systemctl start sshd.socket
```

## zsh Setup

During the user creation, we already installed the *zsh* shell. We have also activated a basic setup at first login by the user.

In this section, we will be installing my variation of zprezto package to manage *zsh* configurations.

First install the main zprezto package:

```
$ git clone --recursive https://github.com/sorin-ionescu/prezto.git "${ZDOTDIR:-$HOME}/.zprezto"
$
$ setopt EXTENDED_GLOB
$ for rcfile in "${ZDOTDIR:-$HOME}"/.zprezto/runcoms/^README.md(.N);
do
    ln -sf "$rcfile" "${ZDOTDIR:-$HOME}/.${rcfile:t}"
done
$
```

Now, We will add my version of prezto to the same git repo.

```
$ cd ~/.zprezto
$ git remote add personal git@github.com:sadanand-singh/My-Zprezto.git
$ git pull personal arch
$ git checkout arch
$ git merge master
```

And we are all setup for using *zsh*!

## gpg Setup

We have already installed the *gnupg* package during the *pacaur* installation. We will first either import our already existing private keys(s) or create one.

Once We have our keys setup, edit keys to change trust level.

Once all keys are setup, we need to gpg-agent configuration file:

```
$ vim ~/.gnupg/gpg-agent.conf
..
enable-ssh-support
default-cache-ttl-ssh 10800
default-cache-ttl 10800
max-cache-ttl-ssh 10800
...
$
```

Also, add following to your *.zshrc* or .“*bash*”*rc* file. If you are using my zprezto setup, you already have this!

```
$ vim ~/.zshrc
...
# set GPG TTY
export GPG_TTY=$(tty)

# Refresh gpg-agent tty in case user switches into an X Session
gpg-connect-agent updatestartuptty /bye >/dev/null

# Set SSH to use gpg-agent
unset SSH_AGENT_PID
if [ "${gnupg_SSH_AUTH_SOCK_by:-0}" -ne $$ ]; then
  export SSH_AUTH_SOCK="/run/user/$UID/gnupg/S.gpg-agent.ssh"
fi
...
$
```

Now, simply start the following systemd sockets as user:

```
$ systemctl --user enable gpg-agent.socket
$ systemctl --user enable gpg-agent-ssh.socket
$ systemctl --user enable dirmngr.socket
$ systemctl --user enable gpg-agent-browser.socket
$
$ systemctl --user start gpg-agent.socket
$ systemctl --user start gpg-agent-ssh.socket
$ systemctl --user start dirmngr.socket
$ systemctl --user start gpg-agent-browser.socket
```

Finally add your ssh key to ssh agent.

```
$ ssh-add ~/.ssh/id_ed25519
```

## User Wallpapers

You can store your own wallpapers at the following location. A good place to get some good wallpapers are KaOS Wallpapers.

```
$ mkdir -p $ $HOME/.local/wallpapers
$ cp SOME_JPEG $HOME/.local/wallpapers/
```

## *conky* Setup

First installed the *conky* package with lua and nvidia support:

```
$ paci conky-lua-nv
```

Then, copy your conky configuration at \$HOME/.config/conky/conky.conf.

```
$ mkdir -p $HOME/.config/conky
# Generate sample conky config file
$ conky -C > $HOME/.config/conky/conky.conf
$
# start conky in background
$ conky &
```

Here, I have also put my simple configuration file:

```
conky.config = {
        background = true,
        use_xft = true,
        xftalpha = 0.2,
        update_interval = 1,
        total_run_times = 0,
        own_window_argb_visual = true,
        own_window = true,
        own_window_type = 'dock',
        own_window_transparent = true,
        own_window_hints = 'undecorated,below,sticky,skip_taskbar,skip_pager',
        double_buffer = true,
        draw_shades = false,
        draw_outline = false,
        draw_borders = false,
        draw_graph_borders = false,
        stippled_borders = 0,
        border_width = 0,
        default_color = 'white',
        default_shade_color = '#000000',
        default_outline_color = '#000000',
        minimum_width = 2500, minimum_height = 3500,
        maximum_width = 2500,
        gap_x = 2980,
        gap_y = 0,
        alignment = 'top_left',
        no_buffers = true,
        uppercase = false,
        cpu_avg_samples = 2,
        net_avg_samples = 2,
        --short_units = true,
        text_buffer_size = 2048,
        use_spacer = 'none',
        override_utf8_locale = true,
        color1 = '#424240',
        color2 = '2a2b2f',
        color3 = '#FF4B4C',--0E87E4
        color4 = '#73bcca',
        own_window_argb_value = 0,
        --own_window_colour = '#000000',
--lua_load rings-v1.2.1.lua
        lua_draw_hook_pre = 'ring_stats',

--lua_load lilas_rings.lua
        lua_draw_hook_post = 'main',
};

conky.text = [[
${goto 200}${voffset 100}${color2}${font Nothing You Could Do:size=50}${time %I:%M}${font Nothing You Could Do:size=20}${time %p}
${goto 185}${voffset 10}${color4}${font Bad Script:size=30}${time %A}
${goto 185}${voffset -35}${font Bad Script:size=18}${time  %d %B, %Y}

${goto -80}${voffset -35}${font Pompiere:size=11}${color 3eafe8}//${color4} CPU: ${execi 1000 cat /proc/cpuinfo | grep 'model name' | sed -e 's/model name.*: //'| uniq | cut -c 19-25} ${color ff3d3d}${hwmon 0 temp 1}°C ${color 3eafe8}//${color4} Load: ${color ff3d3d} ${cpu cpu0}% ${color 3eafe8}// RAM:${color ff3d3d} ${memperc}% / $memmax ${color 3eafe8}//

${goto -80}${voffset -35}${font Pompiere:size=11}${color 3eafe8}//${color4} GPU: ${execi 1000000 nvidia-smi --query-gpu="name,driver_version" --format="csv,noheader" | cut -c 9-18} ${color ff3d3d} ${nvidia temp}°C ${color 3eafe8}//${color4} Load: ${color ff3d3d}${exec nvidia-smi --query-gpu="utilization.gpu" --format="csv,noheader"} ${color 3eafe8}// Free: ${color ff3d3d} ${exec nvidia-smi --query-gpu="memory.free" --format="csv,noheader"} ${color 3eafe8}//

]];
```

## Software Installations

Here is a running list of other common softwares that I install.

```
$ paci spotify tmux tree dropbox thesilver_searcher
$ paci digikam imagemagick
```

I also add the following repository to install the Sublime Text editor. Refer to my previous post for details on setting up Sublime Text.

```
$ curl -O https://download.sublimetext.com/sublimehq-pub.gpg
$ sudo pacman-key --add sublimehq-pub.gpg
$ sudo pacman-key --lsign-key 8A8F901A
$ rm sublimehq-pub.gpg
$
$ echo -e "\n[sublime-text]\nServer = https://download.sublimetext.com/arch/dev/x86_64" | sudo tee -a /etc/pacman.conf
```

Now we can install *sublime-text* as:

```
$ paci sublime-text/sublime-text
```

This brings us to the conclusion of this installation guide. Hope many of you find it useful. Please drop your comments below if you have any suggestions for improvements etc.

**TAGGED IN:** Linux Arch Linux Plasma 5 KDE

Previous post Next post

comments powered by Disqus