Popcorn Linux Installation Guide

This guide presents how to install and run Popcorn Linux in a VirtualBox Virtual Machine (VM) running Debian 7 64bits. The instructions here should be generalizable to other Linux distribution and machines.

Note that a VM with the complete environment resulting from following this guide is available here: `http://bit.ly/2nGz1JY`.

1 Creating the VM

The VM should contain at least two CPUs as Popcorn kernels run on different cores. We will build the kernel and related tools in the virtual machine itself, so a fair amount of disk space is needed: 20GB. Moreover, the more CPUs there is in the VM, the more faster the build will be as we can make use of the parallel building features of make (`-j` flag).

Grab an ISO of Debian 7 and install it. When prompted, do not install a graphical environment. Once the installation is done, install as root the following packages:

```
apt-get update
apt-get install -y sudo build-essential ssh git gawk busybox dropbear udhcpc strace
apt-get install -y libelf-dev binutils-dev kexec-tools mkelfimage autoconf libtool
apt-get install -y automake autotools-dev cu libncurses5-dev vim
```

When prompted if kexec-tools should handle reboots, answer **no**.

2 Downloading Popcorn & related tools sources

In the VM:

```
wget http://www.popcornlinux.org/images/documents/mklinux.tar.gz
wget http://www.popcornlinux.org/images/documents/mklinux-utils.tar.gz
wget http://www.popcornlinux.org/images/documents/mklinux-kexec.tar.gz

mkdir mklinux mklinux-utils mklinux-kexec
tar xf mklinux.tar.gz -C mklinux
tar xf mklinux-utils.tar.gz -C mklinux-utils
tar xf mklinux-kexec.tar.gz -C mklinux-kexec
```

3 Compiling and installing the kernel

Into the `mklinux/` directory, setup the kernel compilation configuration file:

```
cd mklinux
cp configft .config
yes "" | make oldconfig
```

Edit the configuration:

```
make menuconfig
```

Disable these two features:

- Device Drivers › Input device support › Generic input layer › Event debugging
- Device Drivers › Network devices support › Network core driver support › Popcorn shared memory TUN/TAP device driver support

Then, launch the compilation and installation process using the `kinst.sh` script:

```
sudo ./kinst.sh
```

At the end of the compilation process the kernel is installed in /boot. Configure grub so that you will see the printk messages on the serial line of the VM by using the following settings in /etc/default/grub:

```
GRUB_CMDLINE_LINUX_DEFAULT=""
GRUB_CMDLINE_LINUX="console=ttyS0,115200 console=tty0 earlyprintk=ttyS0,115200"
```

Update the grub configuration:

```
sudo update-grub
```

At that point you need to re-instruct kexec *not* to take care of reboots. To do so:

```
sudo dpkg-reconfigure kexec-tools
```

Answers **no** to *Should kexec-tools handle reboots?* and **yes** to *Read GRUB configuration file to load the kernel?*.

Then reboot into the Popcorn kernel. You might need to select it explicitly from grub menu:
Advanced options for Debian GNU/Linux > Debian GNU/Linux, with Linux 3.2.14ft

4 Cpu and memory partitioning configuration

Once you have booted the Popcorn kernel, got to the `mklinux-utils` directory:

```
cd mklinux-utils
```

Compile the C files:

```
make all
```

Then run `generate_all.sh`:

```
./generate_all.sh
```

`generate_all.sh` obtains information about cpus and memory from the currently running Popcorn kernel and generates the boot parameters to be used for launching Popcorn kernels in a multi-kernel setup.
`generate_all.sh` should have created the files `boot_args_i.args` and `boot_args_i.params`, for each cpu `i` of the VM.

Add the following parameters to each `boot_args_i.args` file: `norandmaps vsyscall64=0 vdso=0`

Next, edit once again the boot parameters of the primary kernel:

```
sudo vim /etc/default/grub
```

Modify `GRUB_CMDLINE_LINUX` using the values of `vty_offset` and `mem` found in `boot_args_0.args`, and adding `acpi_irq_nobalance no_ipi_broadcast=1 present_mask=??`, where `??` is the set of cpus you would like the primary kernel to use.

In addition, add the following parameters to `GRUB_CMDLINE_LINUX`:
`norandmaps vsyscall64=0 vdso=0`

For example, to use cpu 0 and 512M of memory:

```
GRUB_CMDLINE_LINUX_DEFAULT="earlyprintk=ttyS0,115200 console=ttyS0,115200 norandmaps
    vsyscall64=0 vdso=0 acpi_irq_nobalance no_ipi_broadcast=1 vty_offset=0x1bc000000
    present_mask=1 mem=512M"
```

Ways to define the CPU partitioning are:

- Only CPU `x`: `present_mask=x`
- The whole CPU range from CPU `x` to `y`: `present_mask=x-y`
- Only CPU `x`, `y` and `z`: `present_mask=x,y,z`

Concerning the memory, the primary kernel should always have access to the first partition of physical memory so just use `mem=xM`, `x` being a numeric value, and `M` can be replaced by `K` or `G`, or nothing to put the unit as bytes.

5 Setting `vty_offset` and `TUN_ADDR`

5.1 `vty_offset`

The value generated for `vty_offset` might be incorrect. It is possible that it is located in a reserved range of the physical memory (it's a physical address), and in that case the secondary kernel will refuse to boot. To perform the check examine the ranges reported by:

```
cat /proc/iomem
```

It `vty_offset` is located in one of the reported ranges, you need to find a free location and place `vty_offset` there. For example if the originally generated `vty_offset` is `0xcc000000`, and in `/proc/iomem` we have this:

```
# ...
000f0000-000fffff : reserved
000f0000-000fffff : System ROM
00100000-3fffffff : System RAM
 01000000-01666e31 : Kernel code
 01666e32-019a00ff : Kernel data
 01a84000-01c27fff : Kernel bss # (2)
bffdf000-bfffffff : reserved # (3)
c0000000-febfffff : PCI Bus 0000:00 # (1)
 fd000000-fdffffff : 0000:00:02.0
 feb80000-febbffff : 0000:00:03.0
 febc0000-febdffff : 0000:00:03.0
 febc0000-febdffff : e1000
 febe0000-febeffff : 0000:00:02.0
 febf0000-febf0fff : 0000:00:02.0
# ...
```

Here we can see that `0xcc000000` falls into the PCI range (1), so we cannot use this `vty_offset`. We need to put it somewhere free, for example between (2) and (3). An example would be `0xb0000000`.

Once you have the final value of `vty_offset`, you need to update:

- The primary kernel command line parameters: in `/etc/default/grub` set the correct value for `vty_offset` in `GRUB_CMDLINE_LINUX` (and do update-grub next then reboot);
- The secondary kernels command line parameters: `mklinux-utils/boot_args_*.args`;
- The tunnel script for the primary kernel: `mklinux-utils/tunnelize.sh`, set the `VTY_ADDR` value;
- The tunnel script for the secondary kernels: `mklinux-utils/custom_ramdisk/packages/01base/tunnelize.sh`, also set the `VTY_ADDR` value.

5.2 `TUN_ADDR`

Running this command will give you an hexadecimal value for `TUN_ADDR`:

```
dmesg | grep virtualTTY | grep phys | sed -rn "s/.*0x.*-(0x.*)$/\1/p"
# This can also give more information than just the hex number
# More precisely, let's take an example: if `dmesg | grep virtualTTY` returns:
# [ 1.109364] virtualTTY: cpu 0, phys 0xb0000000-0xb8000000
# then the value we are searching is 0xb8000000.
```

Edit both tunnelize scripts (`mklinux-utils/tunnelize.sh` and `mklinux-utils/custom_ramdisk/packages/01base/tunnelize.sh`) to set the variable `TUN_ADDR`.

6 Create a ramdisk for the secondary kernels

Next we need to create a custom initramfs that will constitute the root file system of the secondary kernels. That's because the secondary kernels do not have access to the devices of the machine and can therefore not access any file-system located on a disk. Go to the sub-directory `custom_ramdisk` of `mklinux-utils` and run the `make_image.sh` script as follows:

```
cd mklinux-utils/custom_ramdisk
sudo ./make_image.sh ../
```

This should complete without errors and create the file `custom-initramfs.cpio.gz` in the current directory. Copy this file to `mklinux-utils/ramdisk.img`:

```
cp ./custom-initramfs.cpio.gz ../ramdisk.img
```

The created initramfs contains a basic Linux system. Basic utilities are provided by busybox (including init). The initramfs creation scripts in the `custom_ramdisk` directory create an initramfs from scratch and is reasonably simple, so that you can understand it and modify it for your needs.

The basic system utilities are set up in `custom_ramdisk/packages/01base/base.sh`. Have a look at it to see how the initramfs is created and works. In short, the initramfs creation scripts create `custom_ramdisk/image/`, populate it with the needed files (found in `custom_ramdisk/packages/`) , and compress it.

The scripts in `custom_ramdisk/` offer the possibility to copy software from the primary Linux installation to the initramfs of the secondary, including all needed dynamic libraries. To add new software, create a package directory and an installation script in `custom_ramdisk/packages/`. Look at the existing packages to figure out how to do that. Packages can also be blacklisted in `custom_ramdisk/packages/blacklist`.

Note that busybox uses a sysvinit-style init, which use `/etc/inittab` and `/etc/init.d/rc.S`, so you can start new services and execute programs at boot time by modifying those files in the installation script of your new package. Have a look at `custom_ramdisk/packages/dropbear.sh` to see how dropbear is set up to start after boot.

7 Setup kexec

Secondary Popcorn kernels are booted using a modified version of kexec, found in the mklinux-kexec git repository. We first need to install it on the primary:

```
cd ~/mklinux-kexec/
./bootstrap
./configure
make
build/sbin/kexec --version
```

The last command should say the kexec has been compiled from git today. Next, backup the old kexec and install the new one:

```
sudo mv /sbin/kexec /sbin/kexec.old
sudo cp build/sbin/kexec /sbin/kexec
```

Once kexec is installed we need to copy the Popcorn kernel binary to the `mklinux-utils` folder, using the script `create_elf.sh`:

```
cd mklinux-utils
./create_elf.sh ../mklinux/vmlinux
```

8 Boot a secondary kernel

First launch the `tunnelize.sh` script then run the `mklinux_boot.sh` script, passing as parameters the content of `boot_args_i.param`, `i` corresponding to the kernel you want to boot.

```
sudo ./tunnelize.sh
sudo ./mklinux_boot.sh `cat boot_args_1.param`
```

The first secondary kernel is generally assigned the 10.1.2.2 IP address. You can ping it to check everything worked correctly, and also ssh using the id/password combo `root/passwd`:

```
ping 10.1.2.2
ssh root@10.1.2.2
```

You might get a crash dump when the secondary kernel boots with such an error message:

```
Program kexec tried to access /dev/mem between X->Y.
```

To solve this, edit `boot_args_i.args` and modify the kernel / ramdisk physical location in memory not to fall into the range `X->Y`.