Linux Lab v0.4-rc2 Manual

TinyLab Community | Tinylab.org

May 1, 2020

目录

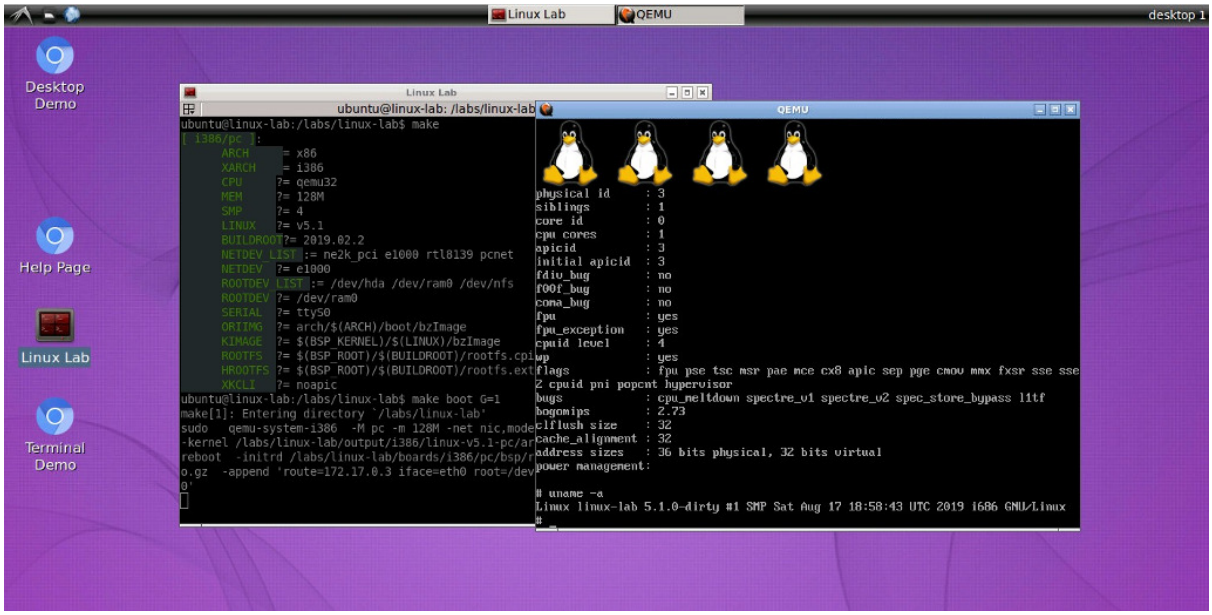
Subscribe Wechat： 

1. Linux Lab Overview

1.1 Project Introduction

This project aims to create a Qemu-based Linux development Lab to easier the learning, development and testing of Linux Kernel.

Linux Lab is open source with no warranty – use at your own risk.



1.2 Project Homepage

- Homepage
  - http://tinylab.org/linux-lab/
- Repository
  - https://gitee.com/tinylab/linux-lab
  - https://github.com/tinyclub/linux-lab

Related Projects:

- Cloud Lab
  - Linux Lab Running Environment Manager
  - http://tinylab.org/cloud-lab
- Linux 0.11 Lab
  - Learning Linux 0.11

  - http://tinylab.org/linux-0.11-lab

- CS630 Qemu Lab
  - Learning X86 Linux Assembly
  - http://tinylab.org/cs630-qemu-lab

1.3 Demonstration

1.3.1 Basic Operations

- Basic Usage
- Learning Uboot
- Learning Assembly
- Boot ARM Ubuntu 18.04 on Vexpress-a9 board
- Boot Linux v5.1 on ARM64/Virt board
- Boot Riscv32/virt and Riscv64/virt boards

1.3.2 Cool Operations

- One command of testing a specified kernel feature
- One command of testing multiple specified kernel modules
- Batch boot testing of all boards
- Batch testing the debug function of all boards

1.4 Project Functions

Now, Linux Lab becomes an intergrated Linux learning, development and testing environment, it supports:

Items	Description
Boards	Qemu based, 8+ main Architectures, 15+ popular boards
Components	Uboot, Linux / Modules, Buildroot, Qemu, Linux v2.6.10 ~ 5.x supported
Prebuilt	All of above components has been prebuilt
Rootfs	Support include initrd, harddisk, mmc and nfs, Debian availab for ARM
Docker	Cross toolchains available in one command, external ones configurable
Acess	Access via web browsers, available everywhere via web vnc or web ssh
Network	Builtin bridge networking, every board has network (except Raspi3)

Items	Description
Boot	Support serial port, curses (ssh friendly) and graphic booting
Testing	Support automatic testing via `make test` target
Debugging	debuggable via `make debug` target

Continue reading for more features and usage.

1.5 Project History

1.5.1 Project Origins

About 9 years ago, a tinylinux proposal: Work on Tiny Linux Kernel accepted by embedded linux foundation, therefore I have worked on this project for serveral months.

1.5.2 Problems Solved

During the project cycle, several scripts written to verify if the adding tiny features (e.g. gc-sections) breaks the other kernel features on the main cpu architectures.

These scripts uses qemu-system-ARCH as the cpu/board simulator, basic boot+function tests have been done for ftrace+perf, accordingly, defconfigs, rootfs, test scripts have been prepared, at that time, all of them were simply put in a directory, without a design or holistic consideration.

1.5.3 Project Born

They have slept in my harddisk for several years without any attention, untill one day, docker and novnc came to my world, at first, Linux 0.11 Lab was born, after that, Linux Lab was designed to unify all of the above scripts, defconfigs, rootfs and test scripts.

2. Linux Lab Installation

2.1 Hardware and Software Requirement

Linux Lab is a full embedded Linux development system, it needs enough calculation capacity and disk & memory storage space, to avoid potential extension issues, here is the recommended configuration:

Hardware	Requirement	Description
Processor	X86_64, > 1.5GHz	Must choose 64bit X86 while using virtual machine
Disk	>= 50G	System (25G), Docker Images(~5G), Linux Lab (20G)
Memory	>= 4G	Lower than 4G may have many unpredictable exceptions

If often use, please increase disk storage to 100G~200G and memory storage to 8G.

And here is a list for verified operating systems for references:

OS	System&Kernel Version	Docker version	Others
Ubuntu	16.04 + 4.4	18.09.4	terminator
Ubuntu	18.04 + 5.0/4.15	18.09.4	Linux v5.3 has issue

Some engineers have run CentOS，Windows 10 and Mac OSX, welcome reply this issue to share yours, for example:

```
$ tools/docker/env.sh
System: Ubuntu 16.04.6 LTS
Linux: 4.4.0-176-generic
Docker: Docker version 18.09.4, build d14af54
```

2.2 Docker Installation

Docker is required by Linux Lab, please install it at first:

- Linux, Mac OSX, Windows 10

  Docker CE

- older Windows (include some older Windows 10)

  Install Ubuntu via Virtualbox or Vmware Virtual Machine

Before running Linux Lab, please make sure the following command works without sudo and without any issue:

```
$ docker run hello-world
```

In China, to use docker service normally, please **must** configure one of chinese docker mirror sites, for example:

- Aliyun Docker Mirror Documentation
- USTC Docker Mirror Documentation

More docker related issues, such as download slowly, download timeout and download errors, are cleary documented in the 6.1 section of FAQs.

The other issues, please read the official docker docs.

Notes for Ubuntu Users - doc/install/ubuntu-docker.md

Notes for Windows Users:

- Please make sure your Windows version support docker: Official Docker Documentation
- Linux Lab only tested with 'Git Bash' in Windows, please must use with it
  - After installing Git For Windows, "Git Bash Here" will come out in right-button press menu

2.3 Choose a working directory

If installed via Docker Toolbox, please enter into the `/mnt/sda1` directory of the `default` system on Virtualbox, otherwise, after poweroff, the data will be lost for the default `/root` directory is only mounted in DRAM.

```
$ cd /mnt/sda1
```

For Linux, please simply choose one directory in `~/Downloads` or `~/Documents`.

```
$ cd ~/Documents
```

For Mac OSX, to compile Linux normally, please create a case sensitive filesystem as the working space at first:

```
$ hdiutil -type SPARSE create -size 60g -fs "Case-sensitive Journaled HFS+" -volname labspace
    labspace.dmg
$ hdiutil attach -mountpoint ~/Documents/labspace -no-browse labspace.dmg
$ cd ~/Documents/labspace
```

Notes: Docker Images, Linux and Buildroot source code require many storage space, please reserve at least 50G for them.

2.4 Download the lab

Use Ubuntu system as an example:

Download cloud lab framework, pull images and checkout linux-lab repository:

```
$ git clone https://gitee.com/tinylab/cloud-lab.git
$ cd cloud-lab/ && tools/docker/choose linux-lab
```

2.5 Run and login the lab

Launch the lab and login with the user and password printed in the console:

```
$ tools/docker/run linux-lab
```

Re-login the lab via web browser:

```
$ tools/docker/vnc linux-lab
```

The other login methods:

```
$ tools/docker/webvnc linux-lab   # The same as tools/docker/vnc
$ tools/docker/webssh linux-lab
$ tools/docker/ssh linux-lab
$ tools/docker/bash linux-lab
```

Summary of login methods:

Login Method	Description	Default User	Where
webvnc	web desktop	ubuntu	anywhere via internet
webssh	web ssh	ubuntu	anywhere via internet

Login Method	Description	Default User	Where
bash	docker bash	ubuntu	localhost
ssh	normal ssh	ubuntu	localhost
vnc	normal vnc	ubuntu	localhost+VNC client

Since vnc clients differs from operating systems, we use webvnc by default to make sure auto login vnc for all systems.

If really want to use local vnc clients, please use the 'IP' and 'Normal Password' printed by `tools/docker/webvnc`, for example, `vinagre` or `remmina` can be used by Ubuntu users.

```
$ tools/docker/vnc linux-lab
...
Please login via VNC Client with:

      IP: 172.17.0.3
    User: 7827c9 (Only for noVNC)
Password: nl7fxd (Normal)
Password: fmkv7w (View)
...

$ sudo apt-get install vinagre

// After connected, input the above 'Normal' Password, the 'View' one is only for students.
$ vinagre 172.17.0.3
```

2.6 Update and rerun the lab

If want a newer version, we **must** back up any local changes at first, for example, save the container:

```
$ tools/docker/commit linux-lab
```

And then update everything:

```
$ tools/docker/update linux-lab
```

If fails, please try to clean up the containers:

```
$ tools/docker/rm-all
```

Or even clean up the whole environments:

```
$ tools/docker/clean-all
```

Then rerun linux lab:

```
$ tools/docker/rerun linux-lab
```

2.7 Quickstart: Boot a board

Issue the following command to boot the prebuilt kernel and rootfs on the default `vexpress-a9` board:

```
$ make boot
```

Login as `root` user without password(password is empty), just input `root` and press Enter:

```
Welcome to Linux Lab

linux-lab login: root

# uname -a
Linux linux-lab 5.1.0 #3 SMP Thu May 30 08:44:37 UTC 2019 armv7l GNU/Linux
```

3. Linux Lab Kickstart

3.1 Using boards

3.1.1 List available boards

List builtin boards:

```
$ make list
[ aarch64/raspi3 ]:
      ARCH     = arm64
      CPU     ?= cortex-a53
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/mmcblk0
[ aarch64/virt ]:
      ARCH     = arm64
      CPU     ?= cortex-a57
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/vda
[ arm/versatilepb ]:
      ARCH     = arm
      CPU     ?= arm926t
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
[ arm/vexpress-a9 ]:
      ARCH     = arm
      CPU     ?= cortex-a9
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
[ i386/pc ]:
      ARCH     = x86
      CPU     ?= i686
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
[ mipsel/malta ]:
      ARCH     = mips
      CPU     ?= mips32r2
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
[ ppc/g3beige ]:
      ARCH     = powerpc
      CPU     ?= generic
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
[ riscv32/virt ]:
      ARCH     = riscv
      CPU     ?= any
      LINUX   ?= v5.0.13
      ROOTDEV ?= /dev/vda
[ riscv64/virt ]:
      ARCH     = riscv
      CPU     ?= any
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/vda
[ x86_64/pc ]:
      ARCH     = x86
      CPU     ?= x86_64
      LINUX   ?= v5.1
      ROOTDEV ?= /dev/ram0
```

ARCH, PLUGIN and FILTER arguments are supported:

```
$ make list ARCH=arm
$ make list PLUGIN=loongson
$ make list FILTER=virt
```

and more:

```
$ make list-board          # only ARCH
$ make list-short          # ARCH and LINUX
$ make list-base           # no plugin
$ make list-plugin         # only plugin
$ make list-full           # everything
```

3.1.2 Choosing a board

By default, the default board: vexpress-a9 is used, we can configure, build and boot for a specific board with BOARD, for example:

```
$ make BOARD=malta
$ make boot
```

If using board, it only works on-the-fly, the setting will not be saved, this is helpful to run multiple boards at the same and not to disrupt each other:

```
$ make board=malta boot
```

This allows to run multi boards in different terminals or background at the same time.

Check the board specific configuration:

```
$ cat boards/arm/vexpress-a9/Makefile
```

3.1.3 Using as plugins

The 'Plugin' feature is supported by Linux Lab, to allow boards being added and maintained in standalone git repositories. Standalone repository is very important to

ensure Linux Lab itself not grow up big and big while more and more boards being added in.

Book examples or the boards with a whole new cpu architecture benefit from such feature a lot, for book examples may use many boards and a new cpu architecture may need require lots of new packages (such as cross toolchains and the architecture specific qemu system tool).

Here maintains the available plugins:

- C-Sky Linux
- Loongson Linux

3.1.4 Configure boards

Every board has its own configuration, some can be changed on demand, for example, memory size, linux version, buildroot version, qemu version and the other external devices, such as serial port, network devices and so on.

The configure method is very simple, just edit it by referring to current values (`boards/<BOARD>/Makefile`), this command open local configuration (`boards/<BOARD>/.labconfig`) via vim:

```
$ make local-edit
```

But please don't make a big change once, we often only need to tune linux version, this command is better for such case:

```
$ make list-linux
v4.12 v4.5.5 v5.0.10 [v5.1]
$ make local-config LINUX=v5.0.10
$ make list-linux
v4.12 v4.5.5 [v5.0.10] v5.1
```

If want to upstream your local changes, please use `board-edit` and `board-config`, otherwise, `local-edit` and `local-config` are preferrable, for they will avoid conflicts while pulling remote updates.

3.2 Build in one command

v0.3+ version add target dependency by default, so, if want to compile a kernel, just run:

```
$ make kernel-build

Or

$ make build kernel
```

It will do everything required, of course, we still be able to run the targets explicitly.

And futher, with the timestamping support, finished targets will not be run again during the late operations, if still want, just clean the stamp and run it again:

```
$ make cleanstamp kernel-build
$ make kernel-build

Or

$ make force-kernel-build
```

To clean all of the stamp files:

```
$ make cleanstamp kernel
```

This function also support uboot, root and qemu.

3.3 Detailed Operations

3.3.1 Downloading

Download board specific package and the kernel, buildroot source code:

```
$ make source APP="bsp kernel root uboot"
Or
$ make source APP=all
Or
$ make source all
```

Download one by one:

```
$ make bsp-source
$ make kernel-source
$ make root-source
$ make uboot-source

Or

$ make source bsp
```

```
$ make source kernel
$ make source root
$ make source uboot
```

3.3.2 Checking out

Checkout the target version of kernel and builroot:

```
$ make checkout APP="kernel root"
```

Checkout them one by one:

```
$ make kernel-checkout
$ make root-checkout

Or

$ make checkout kernel
$ make checkout root
```

If checkout not work due to local changes, save changes and run to get a clean environment:

```
$ make kernel-cleanup
$ make root-cleanup

Or

$ make cleanup kernel
$ make cleanup root
```

The same to qemu and uboot.

3.3.3 Patching

Apply available patches in `boards/<BOARD>/bsp/patch/linux` and `patch/linux/`:

```
$ make kernel-patch

Or

$ make patch kernel
```

3.3.4 Configuration

3.3.4.1 Default Configuration

Configure kernel and buildroot with defconfig:

```
$ make defconfig APP="kernel root"
```

Configure one by one, by default, use the defconfig in `boards/<BOARD>/bsp/`:

```
$ make kernel-defconfig
$ make root-defconfig

Or

$ make defconfig kernel
$ make defconfig root
```

Configure with specified defconfig:

```
$ make B=raspi3
$ make kernel-defconfig KCFG=bcmrpi3_defconfig
$ make root-defconfig KCFG=raspberrypi3_64_defconfig
```

If only defconfig name specified, search boards/ at first, and then the default configs path of buildroot, u-boot and linux-stable respectivly: buildroot/configs, u-boot/configs, linux-stable/arch//configs.

3.3.4.2 Manual Configuration

```
$ make kernel-menuconfig
$ make root-menuconfig

Or

$ make menuconfig kernel
$ make menuconfig root
```

3.3.4.3 Old default configuration

```
$ make kernel-olddefconfig
$ make root-olddefconfig
$ make uboot-olddefconfig

Or

$ make olddefconfig kernel
```

```
$ make olddefconfig root
$ make olddefconfig uboot
```

3.3.5 Building

Build kernel and buildroot together:

```
$ make build APP="kernel root"
```

Build them one by one:

```
$ make kernel-build  # make kernel
$ make root-build    # make root

Or

$ make build kernel
$ make build root
```

3.3.6 Saving

Save all of the configs and rootfs/kernel/dtb images:

```
$ make save APP="kernel root"
$ make saveconfig APP="kernel root"
```

Save configs and images to `boards/<BOARD>/bsp/`:

```
$ make kernel-saveconfig
$ make root-saveconfig
$ make root-save
$ make kernel-save

Or

$ make saveconfig kernel
$ make saveconfig root
$ make save kernel
$ make save root
```

3.3.7 Booting

Boot with serial port (nographic) by default, exit with CTRL+a x, poweroff, reboot or pkill qemu (See poweroff hang):

```
$ make boot
```

Boot with graphic (Exit with CTRL+ALT+2 quit):

```
$ make b=pc boot G=1 LINUX=v5.1
$ make b=versatilepb boot G=1 LINUX=v5.1
$ make b=g3beige boot G=1 LINUX=v5.1
$ make b=malta boot G=1 LINUX=v2.6.36
$ make b=vexpress-a9 boot G=1 LINUX=v4.6.7 // LINUX=v3.18.39 works too
```

Note: real graphic boot require LCD and keyboard drivers, the above boards work well, with linux v5.1, raspi3 and malta has tty0 console but without keyboard input.

vexpress-a9 and virt has no LCD support by default, but for the latest qemu, it is able to boot with G=1 and switch to serial console via the 'View' menu, this can not be used to test LCD and keyboard drivers. XOPTS specify the eXtra qemu options.

```
$ make b=vexpress-a9 CONSOLE=ttyAMA0 boot G=1 LINUX=v5.1
$ make b=raspi3 CONSOLE=ttyAMA0 XOPTS="-serial vc -serial vc" boot G=1 LINUX=v5.1
```

Boot with curses graphic (friendly to ssh login, not work for all boards, exit with ESC+2 quit or ALT+2 quit):

```
$ make b=pc boot G=2 LINUX=v4.6.7
```

Boot with PreBuilt Kernel, Dtb and Rootfs:

```
$ make boot PBK=1 PBD=1 PBR=1
or
$ make boot k=0 d=0 r=0
or
$ make boot kernel=0 dtb=0 root=0
```

Boot with new kernel, dtb and rootfs if exists:

```
$ make boot PBK=0 PBD=0 PBR=0
or
$ make boot k=1 d=1 r=1
or
$ make boot kernel=1 dtb=1 root=1
```

Boot with new kernel and uboot, build them if not exists:

```
$ make boot BUILD="kernel uboot"
```

Boot without Uboot (only `versatilepb` and `vexpress-a9` boards tested):

```
$ make boot U=0
```

Boot with different rootfs (depends on board, check `/dev/` after boot):

```
$ make boot ROOTDEV=/dev/ram      // support by all boards, basic boot method
$ make boot ROOTDEV=/dev/nfs      // depends on network driver, only raspi3 not work
$ make boot ROOTDEV=/dev/sda
$ make boot ROOTDEV=/dev/mmcblk0
$ make boot ROOTDEV=/dev/vda      // virtio based block device
```

Boot with extra kernel command line (XKCLI = eXtra Kernel Command LIne):

```
$ make boot ROOTDEV=/dev/nfs XKCLI="init=/bin/bash"
```

List supported options:

```
$ make list ROOTDEV
$ make list BOOTDEV
$ make list CCORI
$ make list NETDEV
$ make list LINUX
$ make list UBOOT
$ make list QEMU
```

And more `<xxx>-list` are also supported with `list <xxx>`, for example:

```
$ make list features
$ make list modules
$ make list gcc
```

4. Linux Lab Advance

4.1 Using Linux Kernel

4.1.1 non-interactive configuration

A tool named `scripts/config` in linux kernel is helpful to get/set the kernel config options non-interactively, based on it, both of `kernel-getconfig` and `kernel-setconfig` are added to tune the kernel options, with them, we can simply "enable/disable/setstr/setval/getstate" of a kernel option or many at the same time:

Get state of a kernel module:

```
$ make kernel-getconfig m=minix_fs
Getting kernel config: MINIX_FS ...

output/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m
```

Enable a kernel module:

```
$ make kernel-setconfig m=minix_fs
Setting kernel config: m=minix_fs ...

output/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m

Enable new kernel config: minix_fs ...
```

More control commands of `kernel-setconfig` including `y, n, c, o, s, v`:

Option	Description
`y`	build the modules in kernel or enable anther kernel options.
`c`	build the modules as pluginable modules, just like `m`.
`o`	build the modules as pluginable modules, just like `m`.
`n`	disable a kernel option.
`s`	`RTC_SYSTOHC_DEVICE="rtc0"`, set the rtc device to rtc0
`v`	`v=PANIC_TIMEOUT=5`, set the kernel panic timeout to 5 secs.

Operates many options in one command line:

```
$ make kernel-setconfig m=tun,minix_fs y=ikconfig v=panic_timeout=5 s=DEFAULT_HOSTNAME=linux-
    lab n=debug_info
$ make kernel-getconfig o=tun,minix,ikconfig,panic_timeout,hostname
```

4.1.2 using kernel modules

Build all internel kernel modules:

```
$ make modules
$ make modules-install
$ make root-rebuild     // not need for nfs boot
$ make boot
```

List available modules in `modules/`, `boards/<BOARD>/bsp/modules/`:

```
$ make module-list
```

If `m` argument specified, list available modules in `modules/`, `boards/<BOARD>/bsp/modules/` and `linux-stable/`:

```
$ make module-list m=hello
     1  m=hello ; M=$PWD/modules/hello
$ make module-list m=tun,minix
     1  c=TUN ; m=tun ; M=drivers/net
     2  c=MINIX_FS ; m=minix ; M=fs/minix
```

Enable one kernel module:

```
$ make kernel-getconfig m=minix_fs
Getting kernel config: MINIX_FS ...

output/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m

$ make kernel-setconfig m=minix_fs
Setting kernel config: m=minix_fs ...

output/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m

Enable new kernel config: minix_fs ...
```

Build one kernel module (e.g. minix.ko):

```
$ make module M=fs/minix/
Or
$ make module m=minix
```

Install and clean the module:

```
$ make module-install M=fs/minix/
$ make module-clean M=fs/minix/
```

More flexible usage:

```
$ make kernel-setconfig m=tun
$ make kernel x=tun.ko M=drivers/net
$ make kernel x=drivers/net/tun.ko
$ make kernel-run drivers/net/tun.ko
```

Build external kernel modules (the same as internel modules):

```
$ make module m=hello
Or
$ make kernel x=$PWD/modules/hello/hello.ko
```

4.1.3 using kernel features

Kernel features are abstracted in 'feature/linux/, including their configurations patch-set, it can be used to manage both of the out-of-mainline and in-mainline features.

```
$ make feature-list
[ feature/linux ]:
  + 9pnet
  + core
    - debug
    - module
  + ftrace
    - v2.6.36
      * env.g3beige
      * env.malta
      * env.pc
      * env.versatilepb
    - v2.6.37
      * env.g3beige
  + gcs
    - v2.6.36
      * env.g3beige
      * env.malta
      * env.pc
      * env.versatilepb
  + kft
    - v2.6.36
      * env.malta
      * env.pc
  + uksm
    - v2.6.38
```

Verified boards and linux versions are recorded there, so, it should work without any issue if the environment not changed.

For example, to enable kernel modules support, simply do:

```
$ make feature f=module
$ make kernel-olddefconfig
$ make kernel
```

For `kft` feature in v2.6.36 for malta board:

```
$ make BOARD=malta
$ export LINUX=v2.6.36
$ make kernel-checkout
$ make kernel-patch
$ make kernel-defconfig
$ make feature f=kft
$ make kernel-olddefconfig
$ make kernel
$ make boot
```

4.2 Using Uboot Bootloader

Choose one of the tested boards: `versatilepb` and `vexpress-a9`.

```
$ make BOARD=vexpress-a9
```

Download Uboot:

```
$ make uboot-source
```

Checkout the specified version:

```
$ make uboot-checkout
```

Patching with necessary changes, `BOOTDEV` and `ROOTDEV` available, use `flash` by default.

```
$ make uboot-patch
```

Use `tftp`, `sdcard` or `flash` explicitly, should run `make uboot-checkout` before a new `uboot-patch`:

```
$ make uboot-patch BOOTDEV=tftp
$ make uboot-patch BOOTDEV=sdcard
$ make uboot-patch BOOTDEV=flash
```

BOOTDEV is used to specify where to store and load the images for uboot, ROOTDEV is used to tell kernel where to load the rootfs.

Configure:

```
$ make uboot-defconfig
$ make uboot-menuconfig
```

Building:

```
$ make uboot
```

Boot with BOOTDEV and ROOTDEV, use `flash` by default:

```
$ make boot U=1
```

Use `tftp`, `sdcard` or `flash` explicitly:

```
$ make boot U=1 BOOTDEV=tftp
$ make boot U=1 BOOTDEV=sdcard
$ make boot U=1 BOOTDEV=flash
```

We can also change ROOTDEV during boot, for example:

```
$ make boot U=1 BOOTDEV=flash ROOTDEV=/dev/nfs
```

Clean images if want to update ramdisk, dtb and uImage:

```
$ make uboot-images-clean
$ make uboot-clean
```

Save uboot images and configs:

```
$ make uboot-save
$ make uboot-saveconfig
```

4.3 Using Qemu Emulator

Builtin qemu may not work with the newest linux kernel, so, we need compile and add external prebuilt qemu, this has been tested on vexpress-a9 and virt board.

At first, build qemu-system-ARCH:

```
$ make B=vexpress-a9

$ make qemu-download
$ make qemu-checkout
$ make qemu-patch
$ make qemu-defconfig
$ make qemu
$ make qemu-save
```

qemu-ARCH-static and qemu-system-ARCH can not be compiled together. to build qemu-ARCH-static, please enable `QEMU_US=1` in board specific Makefile and rebuild it.

If QEMU and QTOOL specified, the one in bsp submodule will be used in advance of one installed in system, but the first used is the one just compiled if exists.

While porting to newer kernel, Linux 5.0 hangs during boot on qemu 2.5, after compiling a newer qemu 2.12.0, no hang exists. please take notice of such issue in the future kernel upgrade.

4.4 Using Toolchains

The pace of Linux mainline is very fast, builtin toolchains can not keep up, to reduce the maintaining pressure, external toolchain feature is added. for example, ARM64/virt, CCVER and CCPATH has been added for it.

List available prebuilt toolchains:

```
$ make gcc-list
```

Download, decompress and enable the external toolchain:

```
$ make gcc
```

Switch compiler version if exists, for example:

```
$ make gcc-switch CCORI=internal GCC=4.7

$ make gcc-switch CCORI=linaro
```

If not external toolchain there, the builtin will be used back.

If no builtin toolchain exists, please must use this external toolchain feature, currently, aarch64, arm, riscv, mipsel, ppc, i386, x86_64 support such feature.

GCC version can be configured in board specific Makefile for Linux, Uboot, Qemu and Root, for example:

```
GCC[LINUX_v2.6.11.12] = 4.4
```

With this configuration, GCC will be switched automatically during defconfig and compiling of the specified Linux v2.6.11.12.

To build host tools, host gcc should be configured too(please specify b=`i386/pc` explicitly):

```
$ make gcc-list b=i386/pc
$ make gcc-switch CCORI=internal GCC=4.8 b=i386/pc
```

4.5 Using Rootfs

Builtin rootfs is minimal, is not enough for complex application development, which requires modern Linux distributions.

Such a type of rootfs has been introduced and has been released as docker image, ubuntu 18.04 is added for arm32v7 at first, more later.

Run it via docker directly:

```
$ docker run -it tinylab/arm32v7-ubuntu
```

Extract it out and run in Linux Lab:

ARM32/vexpress-a9 (user: root, password: root):

```
$ tools/root/docker/extract.sh tinylab/arm32v7-ubuntu arm
$ make boot B=vexpress-a9 U=0 V=1 MEM=1024M ROOTDEV=/dev/nfs ROOTFS=$PWD/prebuilt/fullroot/tmp
    /tinylab-arm32v7-ubuntu
```

ARM64/raspi3 (user: root, password: root):

```
$ tools/root/docker/extract.sh tinylab/arm64v8-ubuntu arm
$ make boot B=raspi3 V=1 ROOTDEV=/dev/mmcblk0 ROOTFS=$PWD/prebuilt/fullroot/tmp/tinylab-
    arm64v8-ubuntu
```

More rootfs from docker can be found:

```
$ docker search arm64 | egrep "ubuntu|debian"
arm64v8/ubuntu    Ubuntu is a Debian-based Linux operating system  25
arm64v8/debian    Debian is a Linux distribution that's composed  20
```

4.6 Debugging Linux and Uboot

4.6.1 Debugging Linux

Compile the kernel with debugging options:

```
$ make feature f=debug
$ make kernel-olddefconfig
$ make kernel
```

Compile with one thread:

```
$ make kernel JOBS=1
```

And then debug it directly:

```
$ make debug
```

It will open a new terminal, load the scripts from `.gdbinits/kernel`, run gdb automatically.

It equals to:

```
$ make debug linux
or
$ make boot DEBUG=linux
```

to automate debug testing:

```
$ make test-debug linux
or
$ make test DEBUG=linux
```

find out the code line of a kernel panic address:

```
$ make kernel-calltrace func+offset/length
```

4.6.2 Debugging Uboot

to debug uboot with `.gdbinits/uboot`:

```
$ make debug uboot
or
$ make boot DEBUG=uboot
```

to automate uboot debug testing:

```
$ make test-debug uboot
or
$ make test DEBUG=uboot
```

4.7 Test Automation

Use `aarch64/virt` as the demo board here.

```
$ make BOARD=virt
```

Prepare for testing, install necessary files/scripts in `system/`:

```
$ make rootdir
$ make root-install
$ make root-rebuild
```

Simply boot and poweroff (See poweroff hang):

```
$ make test
```

Don't poweroff after testing:

```
$ make test TEST_FINISH=echo
```

Run guest test case:

```
$ make test TEST_CASE=/tools/ftrace/trace.sh
```

Run guest test cases (`COMMAND_LINE_SIZE` must be big enough, e.g. 4096, see `cmdline_size` feature below):

```
$ make test TEST_BEGIN=date TEST_END=date TEST_CASE='ls /root,echo hello world'
```

Reboot the guest system for several times:

```
$ make test TEST_REBOOT=2
```

NOTE: reboot may 1) hang, 2) continue; 3) timeout killed, TEST_TIMEOUT=30; 4) timeout continue, TIMEOUT_CONTINUE=1

Test a feature of a specified linux version on a specified board(`cmdline_size` feature is for increase `COMMAND_LINE_SIZE` to 4096):

```
$ make test f=kft LINUX=v2.6.36 b=malta TEST_PREPARE=board-init,kernel-cleanup
```

NOTE: `board-init` and `kernel-cleanup` make sure test run automatically, but `kernel-cleanup` is not safe, please save your code before use it!!

Test a kernel module:

```
$ make test m=hello
```

Test multiple kernel modules:

```
$ make test m=exception,hello
```

Test modules with specified ROOTDEV, nfs boot is used by default, but some boards may not support network:

```
$ make test m=hello,exception TEST_RD=/dev/ram0
```

Run test cases while testing kernel modules (test cases run between insmod and rmmod):

```
$ make test m=exception TEST_BEGIN=date TEST_END=date TEST_CASE='ls /root,echo hello world'
    TEST_PREPARE=board-init,kernel-cleanup f=cmdline_size
```

Run test cases while testing internal kernel modules:

```
$ make test m=lkdtm TEST_BEGIN='mount -t debugfs debugfs /mnt' TEST_CASE='echo EXCEPTION ">" /
    mnt/provoke-crash/DIRECT'
```

Run test cases while testing internal kernel modules, pass kernel arguments:

```
$ make test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

Run test without feature-init (save time if not necessary, FI=FEATURE_INIT):

```
$ make test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION' FI=0
Or
$ make raw-test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

Run test with module and the module's necessary dependencies (check with `make kernel -menuconfig`):

```
$ make test m=lkdtm y=runtime_testing_menu,debug_fs lkdtm_args='cpoint_name=DIRECT cpoint_type
    =EXCEPTION'
```

Run test without feature-init, boot-init, boot-finish and no TEST_PREPARE:

```
$ make boot-test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

Test a kernel module and make some targets before testing:

```
$ make test m=exception TEST=kernel-checkout,kernel-patch,kernel-defconfig
```

Test everything in one command (from download to poweroff, see poweroff hang):

```
$ make test TEST=kernel,root TEST_PREPARE=board-init,kernel-cleanup,root-cleanup
```

Test everything in one command (with uboot while support, e.g. vexpress-a9):

```
$ make test TEST=kernel,root,uboot TEST_PREPARE=board-init,kernel-cleanup,root-cleanup,uboot-
    cleanup
```

Test kernel hang during boot, allow to specify a timeout, timeout must happen while system hang:

```
$ make test TEST_TIMEOUT=30s
```

Test kernel debug:

```
$ make test DEBUG=1
```

4.8 File Sharing

To transfer files between Qemu Board and Host, three methods are supported by default:

4.8.1 Install files to rootfs

Simply put the files with a relative path in `system/`, install and rebuild the rootfs:

```
$ mkdir system/root/
$ touch system/root/new_file
$ make root-install
$ make root-rebuild
$ make boot
```

4.8.2 Share with NFS

Boot the board with `ROOTDEV=/dev/nfs`:

```
$ make boot ROOTDEV=/dev/nfs
```

Host:

```
$ make env-dump VAR=ROOTDIR
ROOTDIR="/labs/linux-lab/boards/<BOARD>/bsp/root/<BUILDROOT_VERSION>/rootfs"
```

4.8.3 Transfer via tftp

Using tftp server of host from the Qemu board with the `tftp` command.

Host:

```
$ ifconfig br0
inet addr:172.17.0.3  Bcast:172.17.255.255  Mask:255.255.0.0
$ cd tftpboot/
$ ls tftpboot
kft.patch kft.log
```

Qemu Board:

```
$ ls
kft_data.log
$ tftp -g -r kft.patch 172.17.0.3
$ tftp -p -r kft.log -l kft_data.log 172.17.0.3
```

Note: while put file from Qemu board to host, must create an empty file in host firstly. Buggy?

4.8.4 Share with 9p virtio

To enable 9p virtio for a new board, please refer to qemu 9p setup. qemu must be compiled with `--enable-virtfs`, and kernel must enable the necessary options.

Reconfigure the kernel with:

```
CONFIG_NET_9P=y
CONFIG_NET_9P_VIRTIO=y
CONFIG_NET_9P_DEBUG=y (Optional)
CONFIG_9P_FS=y
CONFIG_9P_FS_POSIX_ACL=y
CONFIG_PCI=y
CONFIG_VIRTIO_PCI=y
CONFIG_PCI_HOST_GENERIC=y (only needed for the QEMU Arm 'virt' board)
```

If using `-virtfs` or `-device virtio-9p-pci` option for qemu, must enable the above PCI related options, otherwise will not work:

```
9pnet_virtio: no channels available for device hostshare
mount: mounting hostshare on /hostshare failed: No such file or directory
```

`-device virtio-9p-device` requires less kernel options.

To enable the above options, please simply type:

```
$ make feature f=9pnet
$ make kernel-olddefconfig
```

Docker host:

```
$ modprobe 9pnet_virtio
$ lsmod | grep 9p
9pnet_virtio           17519  0
9pnet                  72068  1 9pnet_virtio
```

Host:

```
$ make BOARD=virt

$ make root-install        # Install mount/umount scripts, ref: system/etc/init.d/S50sharing
$ make root-rebuild

$ touch hostshare/test     # Create a file in host

$ make boot U=0 ROOTDEV=/dev/ram0 PBR=1 SHARE=1

$ make boot SHARE=1 SHARE_DIR=modules   # for external modules development

$ make boot SHARE=1 SHARE_DIR=output/aarch64/linux-v5.1-virt/   # for internal modules
    learning

$ make boot SHARE=1 SHARE_DIR=examples   # for c/assembly learning
```

Qemu Board:

```
$ ls /hostshare/       # Access the file in guest
test
$ touch /hostshare/guest-test   # Create a file in guest
```

Verified boards with Linux v5.1:

boards	Status
aarch64/virt	virtio-9p-device (virtio-9p-pci breaks nfsroot)
arm/vexpress-a9	only work with virtio-9p-device and without uboot booting
arm/versatilepb	only work with virtio-9p-pci
x86_64/pc	only work with virtio-9p-pci
i386/pc	only work with virtio-9p-pci
riscv64/virt	work with virtio-9p-pci and virtio-9p-dev
riscv32/virt	work with virtio-9p-pci and virtio-9p-dev

4.9 Learning Assembly

Linux Lab has added many assembly examples in `examples/assembly`:

```
$ cd examples/assembly
$ ls
aarch64  arm  mips64el  mipsel  powerpc  powerpc64  README.md  x86  x86_64
$ make -s -C aarch64/
Hello, ARM64!
```

4.10 Running any make goals

Linux Lab allows to access Makefile goals easily via `<xxx>-run`, for example:

```
$ make kernel-run help
$ make kernel-run menuconfig

$ make root-run help
$ make root-run busybox-menuconfig

$ make uboot-run help
$ make uboot-run menuconfig
```

`-run` goals allows to run sub-make goals of kernel, root and uboot directly without entering into their own building directory.

5. Linux Lab Development

This introduces how to add a new board for Linux Lab.

5.1 Choose a board supported by qemu

list the boards, use arm as an example:

```
$ qemu-system-arm -M ?
```

5.2 Create the board directory

Use `vexpress-a9` as an example:

```
$ mkdir boards/arm/vexpress-a9/
```

5.3 Clone a Makefile from an existing board

Use `versatilepb` as an example:

```
$ cp boards/arm/versatilebp/Makefile boards/arm/vexpress-a9/Makefile
```

5.4 Configure the variables from scratch

Comment everything, add minimal ones and then others.

Please refer to `doc/qemu/qemu-doc.html` or the online one http://qemu.weilnetz.de/qemu-doc.html.

5.5 At the same time, prepare the configs

We need to prepare the configs for linux, buildroot and even uboot.

Buildroot has provided many examples about buildroot and kernel configuration:

```
buildroot: buildroot/configs/qemu_ARCH_BOARD_defconfig
kernel: buildroot/board/qemu/ARCH-BOARD/linux-VERSION.config
```

Uboot has also provided many default configs:

```
uboot: u-boot/configs/vexpress_ca9x4_defconfig
```

Kernel itself also:

```
kernel: linux-stable/arch/arm/configs/vexpress_defconfig
```

Linux Lab itself also provide many working configs too, the `-clone` target is a good helper to utilize existing configs:

```
$ make list kernel
v4.12 v5.0.10 v5.1
$ make kernel-clone LINUX=v5.1 LINUX_NEW=v5.4
$ make kernel-menuconfig
$ make kernel-saveconfig

$ make list root
2016.05 2019.02.2
$ make root-clone BUILDROOT=2019.02.2 BUILDROOT_NEW=2019.11
$ make root-menuconfig
$ make root-saveconfig
```

Edit the configs and Makefile untill they match our requirements.

```
$ make kernel-menuconfig
$ make root-menuconfig
$ make board-edit
```

The configuration must be put in `boards/<BOARD>/` and named with necessary version info, use `raspi3` as an example:

```
$ make kernel-saveconfig
$ make root-saveconfig
$ ls boards/aarch64/raspi3/bsp/configs/
buildroot_2019.02.2_defconfig  linux_v5.1_defconfig
```

`2019.02.2` is the buildroot version, `v5.1` is the kernel version, both of these variables should be configured in `boards/<BOARD>/Makefile`.

5.6 Choose the versions of kernel, rootfs and uboot

Please use `tag` instead of `branch`, use kernel as an example:

```
$ cd linux-stable
$ git tag
...
v5.0
...
v5.1
..
v5.1.1
v5.1.5
...
```

If want v5.1 kernel, just put a line "LINUX = v5.1" in `boards/<BOARD>/Makefile`.

Or clone a kernel config from the old one or the official defconfig:

```
$ make kernel-clone LINUX_NEW=v5.3 LINUX=v5.1

Or

$ make B=i386/pc
$ pushd linux-stable && git checkout v5.4 && popd
$ make kernel-clone LINUX_NEW=v5.4 KCFG=i386_defconfig
```

If no tag existed, a virtual tag name with the real commmit number can be configured as following:

```
LINUX = v2.6.11.12
LINUX[LINUX_v2.6.11.12] = 8e63197f
```

Linux version specific ROOTFS are also supported:

```
ROOTFS[LINUX_v2.6.12.6]  ?= $(BSP_ROOT)/$(BUILDROOT)/rootfs32.cpio.gz
```

5.7 Configure, build and boot them

Use kernel as an example:

```
$ make kernel-defconfig
$ make kernel-menuconfig
$ make kernel
$ make boot
```

The same to rootfs, uboot and even qemu.

5.8 Save the images and configs

```
$ make root-save
$ make kernel-save
$ make uboot-save

$ make root-saveconfig
$ make kernel-saveconfig
$ make uboot-saveconfig
```

5.9 Upload everything

At last, upload the images, defconfigs, patchset to board specific bsp submodule repository.

Firstly, get the remote bsp repository address as following:

```
$ git remote show origin
* remote origin
  Fetch URL: https://gitee.com/tinylab/qemu-aarch64-raspi3/
  Push  URL: https://gitee.com/tinylab/qemu-aarch64-raspi3/
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (local out of date)
```

Then, fork this repository from gitee.com, upload your changes, and send your pull request.

6. FAQs

6.1 Docker Issues

6.1.1 Speed up docker images downloading

To optimize docker images download speed, please edit `DOCKER_OPTS` in `/etc/default/docker` via referring to `tools/docker/install`.

6.1.2 Docker network conflicts with LAN

We assume the docker network is `10.66.0.0/16`, if not, we'd better change it as following:

```
$ sudo vim /etc/default/docker
DOCKER_OPTS="$DOCKER_OPTS --bip=10.66.0.10/16"

$ sudo vim /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --bip=10.66.0.10/16
```

Please restart docker service and lab container to make this change works:

```
$ sudo service docker restart
$ tools/docker/rerun linux-lab
```

If lab network still not work, please try another private network address and eventually to avoid conflicts with LAN address.

6.1.3 Why not allow running Linux Lab in local host

The full function of Linux Lab depends on the full docker environment managed by Cloud Lab, so, please really never try and therefore please don't complain about why there are lots of packages missing failures and even the other weird issues.

Linux Lab is designed to use pre-installed environment with the docker technology and save our life by avoiding the packages installation issues in different systems, so, Linux Lab would never support local host using even in the future.

6.1.4 Run tools without sudo

To use the tools under `tools` without sudo, please make sure add your account to the docker group and reboot your system to take effect:

```
$ sudo usermod -aG docker $USER
$ newgrp docker
```

6.1.5 Network not work

If ping not work, please check one by one:

- DNS issue

  if `ping 8.8.8.8` work, please check `/etc/resolv.conf` and make sure it is the same as your host configuration.

- IP issue

  if ping not work, please refer to network conflict issue and change the ip range of docker containers.

6.1.6 Client.Timeout exceeded while waiting headers

This means must configure one of the following docker mirror sites:

- Aliyun Docker Mirror Documentation
- USTC Docker Mirror Documentation

Potential methods of configuration in Ubuntu, depends on docker and ubuntu versions:

/etc/default/docker:

```
echo "DOCKER_OPTS=\"\$DOCKER_OPTS --registry-mirror=<your accelerate address>\""
```

/lib/systemd/system/docker.service:

```
ExecStart=/usr/bin/dockerd -H fd:// --bip=10.66.0.10/16 --registry-mirror=<your accelerate
    address>
```

/etc/docker/daemon.json:

```
{
    "registry-mirrors": ["<your accelerate address>"]
}
```

Please restart docker service after change the accelerate address:

```
$ sudo service docker restart
```

For the other Linux systems, Windows and MacOS System, please refer to Aliyun Mirror Speedup Document.

6.1.7 Restart Linux Lab after host system shutdown or reboot

If want to restore the installed softwares and related configurations, please save the container manually:

```
$ tools/docker/commit linux-lab
```

After host system (include virtual machine) shutdown or reboot, you can restart the lab via the "Linux Lab" icon on the desktop, or just like before, issue this command:

```
$ tools/docker/run linux-lab
```

Current implementation doesn't support the direct 'docker start' command, please learn it.

If the above methods still not restart the lab, please refer to the methods mentioned in the 6.3.9 section.

If resume from a suspended host system, the lab will restore automatically, no need to do anything to restart it, just use one of the 4 login methods mentioned in the 2.4 section, for example, start a web browser to connect it:

```
$ tools/docker/vnc linux-lab
```

6.2 Qemu Issues

6.2.1 Why kvm speedding up is disabled

kvm only supports both of `qemu-system-i386` and `qemu-system-x86_64` currently, and it also requires the cpu and bios support, otherwise, you may get this error log:

```
modprobe: ERROR: could not insert 'kvm_intel': Operation not supported
```

Check cpu virtualization support, if nothing output, then, cpu not support virtualization:

```
$ cat /proc/cpuinfo | egrep --color=always "vmx|svm"
```

If cpu supports, we also need to make sure it is enabled in bios features, simply reboot your computer, press 'Delete' to enter bios, please make sure the 'Intel virtualization technology' feature is 'enabled'.

6.2.2 Poweroff hang

Both of the `poweroff` and `reboot` commands not work on these boards currently (LINUX=v5.1):

- mipsel/malta (exclude LINUX=v2.6.36)
- aarch64/raspi3
- arm/versatilepb

System will directly hang there while running `poweroff` or `reboot`, to exit qemu, please pressing `CTRL+a x` or using `pkill qemu`.

To test such boards automatically, please make sure setting `TEST_TIMEOUT`, e.g. `make test TEST_TIMEOUT=50`.

Welcome to fix up them.

6.2.3 How to exit qemu

Where	How
Serial Port Console	`CTRL+a x`
Curses based Graphic	`ESC+2 quit` Or `ALT+2 quit`
X based Graphic	`CTRL+ALT+2 quit`
Generic Methods	`poweroff`, `reboot`, `kill`, `pkill`

6.2.4 Boot with missing sdl2 libraries failure

That's because the docker image is not updated, just rerun the lab (please must not use `tools/docker/restart` here for it not using the new docker image):

```
$ tools/docker/pull linux-lab
$ tools/docker/rerun linux-lab

Or

$ tools/docker/update linux-lab
```

With `tools/docker/update`, every docker images and source code will be updated, it is preferred.

6.3 Environment Issues

6.3.1 NFS/tftpboot not work

If nfs or tftpboot not work, please run `modprobe nfsd` in host side and restart the net services via `/configs/tools/restart-net-servers.sh` and please make sure not use `tools/docker/trun`.

6.3.2 How to switch windows in vim

`CTRL+w` is used in both of browser and vim, to switch from one window to another, please use `CTRL+Left` or `CTRL+Right` key instead, Linux Lab has remapped `CTRL+Right` to `CTRL+w` and `CTRL+Left` to `CTRL+p`.

6.3.3 How to delete typo in shell command line

Long keypress not work in novnc client currently, so, long `Delete` not work, please use `alt+delete` or `alt+backspace` instead, more tips:

Function	Vim	Bash
begin/end	`^/$`	`Ctrl + a/e`
forward/backward	`w/b`	`Ctrl + Home/end`
cut one word backword	`db`	`Alt + Delete/backspace`
cut one word forward	`dw`	`Alt + d`
cut all to begin	`d^`	`Ctrl + u`
cut all to end	`d$`	`Ctrl + k`
paste all cutted	`p`	`Ctrl + y`

6.3.4 Language input switch shortcuts

In order to switch English/Chinese input method, please use CTRL+s shortcuts, it is used instead of CTRL+space to avoid conflicts with local system.

6.3.5 How to tune the screen size

There are tow methods to tune the screen size, one is auto scaling by noVNC, another is pre-setting during launching.

The first one is setting noVNC before connecting.

```
* Press the left sidebar of noVNC web page
* Disconnect
* Enable 'Auto Scaling Mode' via 'Settings -> Scaling Mode: -> Local Scaling -> Apply'
* Connect
```

The second one is setting SCREEN_SIZE while running Linux Lab.

The screen size of lab is captured by xrandr, if not work, please check and set your own, for example:

Get available screen size values:

```
$ xrandr --current
Screen 0: minimum 1 x 1, current 1916 x 891, maximum 16384 x 16384
Virtual1 connected primary 1916x891+0+0 (normal left inverted right x axis y axis) 0mm x 0mm
   1916x891      60.00*+
   2560x1600     59.99
   1920x1440     60.00
   1856x1392     60.00
   1792x1344     60.00
   1920x1200     59.88
   1600x1200     60.00
   1680x1050     59.95
   1400x1050     59.98
   1280x1024     60.02
   1440x900      59.89
   1280x960      60.00
   1360x768      60.02
   1280x800      59.81
   1152x864      75.00
   1280x768      59.87
   1024x768      60.00
   800x600       60.32
   640x480       59.94
```

Before running rm command, please save all of your data, for example, save the container:

```
$ tools/docker/commit linux-lab
```

Choose one and configure it:

```
$ cd /path/to/cloud-lab
$ tools/docker/rm-all
$ SCREEN_SIZE=800x600 tools/docker/run linux-lab
```

If want the default one, please remove the manual setting at first:

```
$ cd /path/to/cloud-lab
$ rm configs/linux-lab/docker/.screen_size
$ tools/docker/rm-all
$ tools/docker/run linux-lab
```

6.3.6 How to work in fullscreen mode

Open the left sidebar, press the 'Fullscreen' button.

6.3.7 How to record video

- Enable recording

  Open the left sidebar, press the 'Settings' button, config 'File/Title/Author/Category/Tags/Description' and enable the 'Record Screen' option.

- Start recording

  Press the 'Connect' button.

- Stop recording

  Press the 'Disconnect' button.

- Replay recorded video

  Press the 'Play' button.

- Share it

  Videos are stored in 'cloud-lab/recordings', share it with help from showdesk.io.

6.3.8 Linux Lab not response

The VNC connection may hang for some unknown reasons and therefore Linux Lab may not response sometimes, to restore it, please press the flush button of web browser or re-connect after explicitly disconnect.

6.3.9 VNC login with failures

If VNC login return "Disconnect timeout", wait a while and press the left 'Connect' button again, otherwise, check as following:

At first, check the containers' status (Up: Ok, Exit: Bad):

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
19a61ba075b5 tinylab/linux-lab "/tools/lab/run" 4 days ago Up 4 days 22/tcp, 5900/tcp linux-lab-21575
75dae89984c9 tinylab/cloud-ubuntu-web "/startup.sh" 8 days ago Up 8 days ....443/tcp cloud-ubuntu-web
```

If the status is 'Exit', that means container may be shutdown or may never up, run it again to resume for the shutdown case:

```
$ tools/docker/run linux-lab
```

Otherwise, check the running logs:

```
$ tools/docker/logs linux-lab
```

If normal, that means the login account and password may have been invalid for some exceptions, please regenerte new account and password with the coming steps:

Note: The `clean` command will remove some containers and data, please do necessary backup before run it, for example, save the container:

```
$ tools/docker/commit linux-lab
```

VNC login fails while using mismatched password, to fix up such issue, please clean up all and rerun it:

```
$ tools/docker/clean linux-lab
$ tools/docker/rerun linux-lab
```

If the above command not work, please try this one (**It will clean more data, please do necessary backup**)

```
$ tools/docker/clean-all
$ tools/docker/rerun linux-lab
```

6.3.10 Ubuntu Snap Issues

Users report many snap issues, please use apt-get instead:

- users can not be added to docker group and break non-root operation.
- snap service exhausts the /dev/loop devices and break mount operation.

6.4 Lab Issues

6.4.1 No working init found

This means the rootfs.ext2 image may be broken, please remove it and try `make boot` again, for example:

```
$ rm boards/aarch64/raspi3/bsp/root/2019.02.2/rootfs.ext2
$ make boot
```

`make boot` command can create this image automatically.

6.4.2 linux/compiler-gcc7.h: No such file or directory

This means using a newer gcc than the one linux kernel version supported, the solution is switching to an older gcc version via `make gcc-switch`, use `i386/pc` board as an example:

```
$ make gcc-list
$ make gcc-switch CCORI=internal GCC=4.4
```

6.4.3 linux-lab/configs: Permission denied

This may happen at `make boot` while the repository is cloned with `root` user, please simply update the owner of `cloud-lab/` directory:

```
$ cd /path/to/cloud-lab
$ sudo chown $USER:$USER -R ./
$ tools/docker/rerun linux-lab
```

To make a consistent working environment, Linux Lab only support using as general user: 'ubuntu'.

6.4.4 scripts/Makefile.headersinst: Missing UAPI file

This means MAC OSX not use Case sensitive filesystem, create one using `hdiutil` or `Disk Utility` yourself:

```
$ hdiutil create -type SPARSE -size 60g -fs "Case-sensitive Journaled HFS+" -volname labspace
    labspace.dmg
$ hdiutil attach -mountpoint ~/Documents/labspace -no-browse labspace.dmg
$ cd ~/Documents/labspace
```

7. Contact and Sponsor

Our contact wechat is **tinylab**, welcome to join our user & developer discussion group.

Contact us and Sponsor via wechat:

Figure 1: contact-sponsor