Inhaltsverzeichnis

Raspberry Pi 4 emulieren mit QEMU

Quasi als moderisierte Variante der ersten Ausgabe hier nun eine Version, die ohne spezielle Modifikation des Image auskommt:

Wir holen uns ein 64-Bit Raspberry Pi OS Image von der |offiziellen Downloadseite und merken uns die Kernel- Version.

Zum Neu-Kompilieren des Kernels installieren wir die notwendigen Tools

```
sudo apt install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu \
qemubuilder qemu-system-gui qemu-system-arm qemu-utils qemu-system-data
qemu-system \
bison flex guestfs-tools libssl-dev telnet xz-utils
```

dann basteln wir uns einen Arbeitsordner und entpacken das Image

```
mkdir ~/rpi-emu
cd ~/rpi-emu
cp ~/Downloads/2024-07-04-raspios-bookworm-arm64-lite.img.xz ./
unxz 2024-07-04-raspios-bookworm-arm64-lite.img.xz
```

Zum Besorgen der passenden Kernel- Sources von www.kernel.org und dem Kompilieren speichern wir das folgende Script als `build-qemu-kernel.sh` und führen es dann aus:

build-qemu-kernel.sh

```
#!/bin/bash
VERSION=6.6.49

wget https://cdn.kernel.org/pub/linux/kernel/v${VERSION//.*/.x}/linux-
${VERSION}.tar.xz
tar -xvJf linux-${VERSION}.tar.xz

cd linux-${VERSION}

ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make defconfig
ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make kvm_guest.config
ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make -j8

cp arch/arm64/boot/Image ../kernel
cd ..
```

```
chmod +x build-qemu-kernel.sh
./build-qemu-kernel.sh
```

das Image liegt nun als `kernel` auf der Platte

Raspberry Pi OS Image anpassen

Damit wir das Raspberry Pi OS Image mit QEMU nutzen können, müssen wir es zuerst noch etwas anpassen.

Image vergrößern

Ein normaler Raspberry Pi vergrößert beim ersten Start automatisch das Dateisystem auf den auf der SD-Karte verfügbaren Speicherplatz. Da dies hier nicht automatisch passiert, müssen wir das Image manuell vergrößern, wobei wir es auch gleichzeitig in eine neue Datei speichern.

Im folgenden speichern wir das neue Image als disk.img und vergrößern es um 20Gb

```
cp 2024-07-04-raspios-bookworm-arm64-lite.img disk.img
truncate -s +20G disk.img
sudo virt-resize --expand /dev/sda2 2024-07-04-raspios-bookworm-arm64-
lite.img disk.img
```

ACHTUNG: Das hier angegebene /dev/sda2 bezieht sich auf die Partition 2 in dem Image. Bitte nicht mit eine ggf. vorhandenen sda2 Partition auf dem eigenen Rechner verwechseln.

Konfiguration von Benutzer und Passwort und aktivieren von SSH

Bei den neueren Raspberry Pi OS Images gibt es keinen Standardbenutzer mehr. Darum müssen wir das Image bearbeiten, um diesen zu erstellen.

Hierzu lassen wir uns zuerst die Partitionen im Image anzeigen.

```
fdisk -l disk.img

Festplatte disk.img: 22,58 GiB, 24243077120 Bytes, 47349760 Sektoren
Einheiten: Sektoren von 1 * 512 = 512 Bytes
Sektorgröße (logisch/physikalisch): 512 Bytes / 512 Bytes
E/A-Größe (minimal/optimal): 512 Bytes / 512 Bytes
Festplattenbezeichnungstyp: dos
Festplattenbezeichner: 0xfb33757d

Gerät      Boot  Anfang     Ende Sektoren Größe Kn Typ
disk.img1          8192  1056767  1048576  512M  c W95 FAT32 (LBA)
disk.img2       1056768 47349375 46292608 22,1G 83 Linux
```

Von der Ausgabe benötigen wir den Anfang von disk.img1 (hier 8192) und die Sektorgröße (hier 512).

Diese beiden Werte multiplizieren wir, um den offset zum Einbinden der Partition zu erhalten. 8192 * 512 = 4194304

Anschließend erstellen wir uns ein Verzeichnis und mounten die Partition aus dem Image unter

Angabe des eben berechneten offset in dieses.

Achtung: Bitte unbedingt auf die korrekte Angabe des offset achten.

Partition aus dem Image mounten

```
mkdir mnt
sudo mount -o loop,offset=4194304 disk.img ./mnt
```

Nun erstellen wir eine Datei userconf.txt in der gemounteten Partition. Diese Datei beinhaltet den Benutzernamen und das Passwort für den Login.

Das Passwort wird interaktiv abgefragt.

```
PW=$(openssl passwd -6)
echo "pi:$PW" | sudo tee mnt/userconf.txt
```

Damit auch SSH in unserem emulierten System aktiviert wird, erstellen wir zudem eine ssh Datei.

```
sudo touch mnt/ssh
```

Anschließend unmounten wir die Partion vom Image wieder.

```
sudo umount mnt
```

Starten des emulierten Systems

Nun haben wir alles soweit vorbereitet, um das Emulierte Raspberry Pi OS zu starten.

Um den Start zu vereinfachen, erstellen wir zunächst ein kurzes Startskript `rpi-emu-start.sh`.

rpi-emu-start.sh

```
#!/bin/bash
IMAGE=disk.img    # Image
CPU_CORES=4       # CPU-Kerne (bis zu 8)
RAM_SIZE=8G       # Größe des Arbeitsspeichers
SSH_PORT=2222     # Lokaler Port für den SSH-Zugriff
MONITOR_PORT=5555 # Lokaler Port für die QEMU Monitor Konsole
ARGS=             # Zusätzliche Argument (-nographic um ohne grafisches Fenster zu starten)

qemu-system-aarch64 -machine virt -cpu cortex-a72 \
  -smp ${CPU_CORES} -m ${RAM_SIZE} \
  -kernel kernel \
  -append "root=/dev/vda2 rootfstype=ext4 rw panic=0 console=ttyAMA0" \
  -drive format=raw,file=${IMAGE},if=none,id=hd0,cache=writeback \
  -device virtio-blk,drive=hd0,bootindex=0 \
  -netdev user,id=mynet,hostfwd=tcp::${SSH_PORT}-:22 \
  -device virtio-net-pci,netdev=mynet \
```

```
  -monitor telnet:127.0.0.1:${MONITOR_PORT},server,nowait \
  $ARGS
```

Dieses Skript machen wir ausführbar und starten es anschließend.

```
chmod +x rpi-emu-start.sh
./rpi-emu-start.sh
```

Der erste Start dauert noch etwas länger.

Login in das System

Da wir den SSH-Server aktiviert und den SSH-Port durchgereicht haben, können wir per SSH wie folgt auf unser emuliertes System zugreifen:

```
ssh -p 2222 pi@localhost
```

Es empfiehlt sich, nach dem ersten kompletten Durchlauf das Image einmal zu kopieren, um so immer eine fertige, aber noch jungfräuliche Installation zu haben, die man dann wieder für weitere Versuche kopieren kann.

Monitor-Konsole

Die QEMU Monitor-Konsole kann per telnet localhost 5555 erreicht werden. Hierrüber ist es beispielsweise möglich Befehle an die QEMU Instanz zu senden.

Qemu mit Display

Und wenn man das obige Beispiel erfolgreich zum Laufen bekommen hat, denkt man, jetzt auch noch mit emulierter GUI kann ja nicht schwer sein. Denkste...

Nach drei Tagen Googlen, Batch- Files hin und her kreuzen und endlosen (und erfolglosen..) Testläufen kam dann endlich folgendes dabei raus: [1)]

create_raspi_vm.sh

```
#!/bin/bash

## Creates and runs a virtual machine running Raspberry Pi OS Desktop
or Lite with a GUI.
## Read it's documentation at
https://gist.github.com/emendir/922d6914a1705ed2e8e4e96db726c422
## Developed based on
https://gist.github.com/cGandom/23764ad5517c8ec1d7cd904b923ad863
```

```
## check the parameters
if [ ! -f "$1" ]
  then
    echo "No valid raspberry pi os image supplied"
    echo "Usage: build-qemu-kernel.sh origial_image_file kernel_version
domain_name"
    exit 1
fi
if [ -z "$2" ]
  then
    echo "Version missing"
    echo "Usage: build-qemu-kernel.sh origial_image_file kernel_version
domain_name"
    exit 1
fi

if [ -z "$3" ]
  then
    echo "domainname missing"
    echo "Usage: build-qemu-kernel.sh origial_image_file kernel_version
domain_name"
    exit 1
fi




# Directory in which the VM's disk and linux kernel images will be
stored
#WORK_DIR=$(pwd)/$VM_NAME
WORK_DIR=/opt/VirtualMachines/$VM_NAME
# create some variables
RPOS_LOCATION="$1"
VERSION="$2"
VM_NAME="$3" # name of the virtual machine registered in virt-manager


## Generate the password hash using the following command:
# openssl passwd -6
PASSWORD_HASH='$6$2mB/106BUYDnL6oT$xPwTaarKh624nsI95IwR.U8xKBup1mbJHZxa
Sii9BcHuq2q2lSfc28LVlm1kPnd2fj/3YdIwoGhCDIR2Y0//Q0'  # this value is
for the password 'raspberry'
USERNAME='pi'


# Download URL of the Linux kernel used
KERNEL_URL=https://cdn.kernel.org/pub/linux/kernel/v${VERSION//.*/.x}/l
inux-${VERSION}.tar.xz
```

```
# Download URL of the Raspberry Pi OS image to use (the following
options have been tested with the above kernel)
#
RPOS_IMAGE_URL=https://downloads.raspberrypi.com/raspios_lite_arm64/ima
ges/raspios_lite_arm64-2024-07-04/2024-07-04-raspios-bookworm-arm64-
lite.img.xz
RPOS_IMAGE_URL=https://downloads.raspberrypi.com/raspios_full_arm64/ima
ges/raspios_full_arm64-2024-07-04/2024-07-04-raspios-bookworm-arm64-
full.img.xz

# Flags to force rebuilding kernel or Raspberry Pi OS images
# RECREATE_RPOS_IMAGE implies REBUILD_KERNEL
RECREATE_RPOS_IMAGE=false   # CAREFULL: will delete any existing image
REBUILD_KERNEL=false         # will reuse existing kernel build files
if found, delete $WORK_DIR/linux-kernel to start from scratch

# amount by which to resize the the VM's Raspberry Pi OS disk image
# (requires manually running `raspi-config nonint do_expand_rootfs` to
apply)
RPOS_IMAGE_EXTRA_SPACE=6G

# location of the VM's Raspberry Pi OS disk image
RPOS_IMAGE=$WORK_DIR/${VM_NAME}_vm_disk.img


# Setup mounting directories to edit the Raspberry Pi OS image
RPI_BOOT_MNT=$WORK_DIR/rpi_boot
RPI_ROOT_MNT=$WORK_DIR/rpi_root

if ! [ -e $WORK_DIR ];then
  sudo mkdir -p $WORK_DIR
fi
sudo chmod -R a+rwx $WORK_DIR
if ! [ -d $RPI_BOOT_MNT ];then
  mkdir $RPI_BOOT_MNT
fi
if ! [ -d $RPI_ROOT_MNT ]; then
  mkdir $RPI_ROOT_MNT
fi

cd $WORK_DIR || exit 1

## Setup disk image for VM with Raspberry Pi OS pre-installed
RECREATED_RPOS_IMAGE=false
if $RECREATE_RPOS_IMAGE || ! [ -e $RPOS_IMAGE ];then
  RECREATED_RPOS_IMAGE=true

  # delete any old images
  if [ -e $RPOS_IMAGE ];then
    rm $RPOS_IMAGE
  fi
```

```
  ## Download and resize Raspberry Pi OS image
  #wget $RPOS_IMAGE_URL -O $RPOS_IMAGE.xz
  cp $RPOS_LOCATION ${RPOS_IMAGE}_unsized.xz
  xz -d ${RPOS_IMAGE}_unsized.xz
  cp ${RPOS_IMAGE}_unsized $RPOS_IMAGE
  truncate -s +$RPOS_IMAGE_EXTRA_SPACE $RPOS_IMAGE
  sudo virt-resize --expand /dev/sda2 ${RPOS_IMAGE}_unsized $RPOS_IMAGE
  #qemu-img resize -f raw $RPOS_IMAGE +$RPOS_IMAGE_EXTRA_SPACE
fi

# ensure we can work with the image now
# we'll change its permissions again before running the VM
chown $USER:kvm $RPOS_IMAGE

# calculate image partition details for the boot and root partitions in
the Raspberry Pi OS image

###### ATTENTION: The output text of the 'Sector size' is locate
dependent and must be adjusted to your Country ! :-|

# sector_size=$(fdisk -l $RPOS_IMAGE | grep 'Sector size' | awk '{print
$4}')

sector_size=$(fdisk -l $RPOS_IMAGE | grep -m1 'Sektorgröße'| awk
'{print $3}')
start_sector_boot=$(fdisk -l $RPOS_IMAGE | grep -m1 "^${RPOS_IMAGE}1" |
awk '{print $2}')
start_sector_root=$(fdisk -l $RPOS_IMAGE | grep -m1 "^${RPOS_IMAGE}2" |
awk '{print $2}')
start_sector_bytes_boot=$((sector_size * start_sector_boot))
start_sector_bytes_root=$((sector_size * start_sector_root))

echo "Image sector size: $sector_size"
echo "Boot partition starts at sector $start_sector_boot (byte offset
$start_sector_bytes_boot)"
echo "Root partition starts at sector $start_sector_root (byte offset
$start_sector_bytes_root)"

if $RECREATED_RPOS_IMAGE;then
  ## Preconfigure username & password, enable SSH
  # mount the first partition of the Raspberry Pi OS image (the boot
partition)
  sudo mount -o loop,offset=$start_sector_bytes_boot $RPOS_IMAGE
$RPI_BOOT_MNT
  echo "${USERNAME}:${PASSWORD_HASH}"| sudo tee $RPI_BOOT_MNT/userconf
  sudo touch $RPI_BOOT_MNT/ssh # enable SSH
  sudo umount $RPI_BOOT_MNT
fi
```

```
## Build Linux Kernel
KERNEL_IMAGE=$WORK_DIR/linux-kernel/arch/arm64/boot/Image

REBUILT_KERNEL=false
if $REBUILD_KERNEL || $RECREATED_RPOS_IMAGE || ! [ -e $KERNEL_IMAGE
];then
  REBUILT_KERNEL=true
  # download linux kernel if necessary
  if ! [ -e $KERNEL_IMAGE ];then
    rm -r linux-* # remove old files
    wget $KERNEL_URL -O linux-kernel.tar.xz
    tar xvJf linux-kernel.tar.xz
    rm linux-kernel.tar.xz
    mv linux-* linux-kernel # rename linux-VERSION to known name
  fi

  cd linux-kernel || exit 1
  # create a .config file
  ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make defconfig

  # Use the kvm_guest config as the base defconfig, which is suitable
for qemu
  ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make
kvm_guest.config

  ## manual kernel configuration DON'T DO THIS ON A MACHINE OF
DIFFERENT ARCHITECTURE
  # make menu config
  sed -i 's/# CONFIG_DRM_QXL is not set/CONFIG_DRM_QXL=m/' .config
  sed -i 's/# CONFIG_FB_SIMPLE is not set/CONFIG_FB_SIMPLE=y/' .config
  sed -i 's/# CONFIG_FB_VIRTUAL is not set/CONFIG_FB_VIRTUAL=y/'
.config
  sed -i 's/# CONFIG_NETLINK_DIAG is not set/CONFIG_NETLINK_DIAG=y/'
.config

  # Build the kernel
  ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- make -j8

  ## Install kernel modules into the guest filesystem
  # mount the second partition of the Raspberry Pi OS image (the root
partition)
  sudo mount -o loop,offset=$start_sector_bytes_root $RPOS_IMAGE
$RPI_ROOT_MNT
  # Install kernel modules into the guest filesystem
  ARCH=arm64 CROSS_COMPILE=/bin/aarch64-linux-gnu- sudo make
modules_install INSTALL_MOD_PATH=$RPI_ROOT_MNT
  sudo umount $RPI_ROOT_MNT

  cd .. || exit 1 # return to parent directory
fi
```

```
sudo chown libvirt-qemu:kvm $RPOS_IMAGE



## Run unregistered VM directly using qemu (CLI only)
# qemu-system-aarch64 -machine virt -cpu cortex-a72 -smp 6 -m 4G \
#     -kernel $KERNEL_IMAGE -append "root=/dev/vda2 rootfstype=ext4 rw
panic=0 console=ttyAMA0" \
#     -drive format=raw,file=$RPOS_IMAGE,if=none,id=hd0,cache=writeback
\
#     -device virtio-blk,drive=hd0,bootindex=0 \
#     -netdev user,id=mynet,hostfwd=tcp::2222-:22 \
#     -device virtio-net-pci,netdev=mynet \
#     -monitor telnet:127.0.0.1:5555,server,nowait

## Create registered VM for virt-manager (with graphics!)
echo "
<domain type='qemu'>
  <name>$VM_NAME</name>
  <memory unit='GiB'>4</memory>
  <vcpu placement='static'>6</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>$KERNEL_IMAGE</kernel>
    <cmdline>root=/dev/vda2 rootfstype=ext4 rw panic=0
console=ttyAMA0</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model>cortex-a72</model>
  </cpu>
  <devices>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='$RPOS_IMAGE'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
      <model type='virtio'/>
    </interface>
    <!--<filesystem type='mount' accessmode='mapped'>
      <source dir='$QBM_SOURCE'/>
      <target dir='$SHARED_FS_TAG'/>
    </filesystem>-->
    <serial type='pty'>
      <source path='/dev/pts/4'/>
      <target type='system-serial' port='0'>
        <model name='pl011'/>
      </target>
```

```
      <alias name='serial0'/>
    </serial>
    <controller type='usb' index='0' model='qemu-xhci' ports='15'>
      <address type='pci' domain='0x0000' bus='0x07' slot='0x00'
function='0x0'/>
    </controller>
    <input type='mouse' bus='usb'/>
    <input type='keyboard' bus='usb'/>
    <graphics type='spice' autoport='yes'>
      <listen type='address'/>
      <image compression='off'/>
      <gl enable='no'/>
    </graphics>
    <audio id='1' type='none'/>
    <video>
      <model type='virtio' heads='1' primary='yes'/>
    </video>
  </devices>
</domain>

" > raspi_vm.config
virsh define raspi_vm.config
virsh start "$VM_NAME"



BLACK='\033[0;30m'
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[0;33m'
BLUE='\033[0;34m'
MAGENTA='\033[0;35m'
CYAN='\033[0;36m'
WHITE='\033[0;37m'
NC='\033[0m' # No Color
echo -e "$YELLOW
It is normal for the VM window to display while booting:
'Guest has not initialized the display (yet).'

Wait for a minute, after which the console log-in should display.
In the virt-manager VM window, under the menu _View > Consoles > Serial
1_ you can switch to the text console while the graphics aren't running
yet.

$NC"
```

Emulieren im Docker Container

Die obige Methode (mit Display) scheint zu funktionieren, solange man nicht Besonderheiten vom Betriebssystem erwartet.

Der Versuch aber, per Ansible erst Docker zu installieren und dann darin einen RabbitMQ- Server, endete mit Docker- Netzwerk- Fehlermeldungen beim Container- Start. Offensichtlich reicht es nicht, wie oben getan, den Standard- Kernel-Source zu laden und nur für die QEmu- Grafikausgabe zu modifizieren und zu compilieren - so ein echter Raspi hat da wohl noch ein paar Tweaks mehr.

Als nächster Versuch kam dann „Starten als Qemu in einem Docker Container“.

Das endete aber in einem Debakel: Die Kernel-Images, die man angepasst an QEmu im Netz findet, sind ältere Rapians mit älteren Python- Versionen, und Docker mögen die auch alle nicht.

Die abschliessende Verzweifelungstat war dann, sich die die Raspian- Sourcen zu clonen und dies dann mit oben schon gefundenen QEmu- Anpassungen zu mixen. Viel Arbeit, doch am Ende gab's immer nur einen schwarzen Bildschirm....

Hier der dazugehörige Code, vielleicht findet ja mal einer die Lösung

create_raspi_vm_raspian.sh

```
#!/bin/bash

## Creates and runs a virtual machine running Raspberry Pi OS Desktop
or Lite with a GUI.
## Read it's documentation at
https://gist.github.com/emendir/922d6914a1705ed2e8e4e96db726c422
## Developed based on
https://gist.github.com/cGandom/23764ad5517c8ec1d7cd904b923ad863

## check the parameters
if [ ! -f "$1" ]
  then
    echo "No valid raspberry pi os image supplied"
    echo "Usage: build-qemu-kernel.sh origial_image_file domain_name"
    exit 1
fi

if [ -z "$2" ]
  then
    echo "domainname missing"
    echo "Usage: build-qemu-kernel.sh origial_image_file domain_name"
    exit 1
fi
```

```
# Directory in which the VM's disk and linux kernel images will be
stored
#WORK_DIR=$(pwd)/$VM_NAME
WORK_DIR=/opt/VirtualMachines/$VM_NAME
# create some variables
RPOS_LOCATION="$1"
VM_NAME="$2" # name of the virtual machine registered in virt-manager


## Generate the password hash using the following command:
# openssl passwd -6
PASSWORD_HASH='$6$2mB/106BUYDnL6oT$xPwTaarKh624nsI95IwR.U8xKBup1mbJHZxa
Sii9BcHuq2q2lSfc28LVlm1kPnd2fj/3YdIwoGhCDIR2Y0//Q0'  # this value is
for the password 'raspberry'
USERNAME='pi'


# Git URL of the Raspberry kernel repository used
KERNEL_URL=https://github.com/raspberrypi/linux



# Download URL of the Raspberry Pi OS image to use (the following
options have been tested with the above kernel)
#
RPOS_IMAGE_URL=https://downloads.raspberrypi.com/raspios_lite_arm64/ima
ges/raspios_lite_arm64-2024-07-04/2024-07-04-raspios-bookworm-arm64-
lite.img.xz
RPOS_IMAGE_URL=https://downloads.raspberrypi.com/raspios_full_arm64/ima
ges/raspios_full_arm64-2024-07-04/2024-07-04-raspios-bookworm-arm64-
full.img.xz

# Flags to force rebuilding kernel or Raspberry Pi OS images
# RECREATE_RPOS_IMAGE implies REBUILD_KERNEL
RECREATE_RPOS_IMAGE=false   # CAREFULL: will delete any existing image
REBUILD_KERNEL=false         # will reuse existing kernel build files
if found, delete $WORK_DIR/linux-kernel to start from scratch

# amount by which to resize the the VM's Raspberry Pi OS disk image
# (requires manually running `raspi-config nonint do_expand_rootfs` to
apply)
RPOS_IMAGE_EXTRA_SPACE=6G

# location of the VM's Raspberry Pi OS disk image
RPOS_IMAGE=$WORK_DIR/${VM_NAME}_vm_disk.img


# Setup mounting directories to edit the Raspberry Pi OS image
RPI_BOOT_MNT=$WORK_DIR/rpi_boot
RPI_ROOT_MNT=$WORK_DIR/rpi_root
```

```
if ! [ -e $WORK_DIR ];then
  sudo mkdir -p $WORK_DIR
fi
sudo chmod -R a+rwx $WORK_DIR
if ! [ -d $RPI_BOOT_MNT ];then
  mkdir $RPI_BOOT_MNT
fi
if ! [ -d $RPI_ROOT_MNT ]; then
  mkdir $RPI_ROOT_MNT
fi

cd $WORK_DIR || exit 1

## Setup disk image for VM with Raspberry Pi OS pre-installed
RECREATED_RPOS_IMAGE=false
if $RECREATE_RPOS_IMAGE || ! [ -e $RPOS_IMAGE ];then
  RECREATED_RPOS_IMAGE=true

  # delete any old images
  if [ -e $RPOS_IMAGE ];then
    rm $RPOS_IMAGE
  fi


  ## Download and resize Raspberry Pi OS image
  #wget $RPOS_IMAGE_URL -O $RPOS_IMAGE.xz
  cp $RPOS_LOCATION ${RPOS_IMAGE}_unsized.xz
  xz -d ${RPOS_IMAGE}_unsized.xz
  cp ${RPOS_IMAGE}_unsized $RPOS_IMAGE
  truncate -s +$RPOS_IMAGE_EXTRA_SPACE $RPOS_IMAGE
  sudo virt-resize --expand /dev/sda2 ${RPOS_IMAGE}_unsized $RPOS_IMAGE
  #qemu-img resize -f raw $RPOS_IMAGE +$RPOS_IMAGE_EXTRA_SPACE
fi

# ensure we can work with the image now
# we'll change its permissions again before running the VM
chown $USER:kvm $RPOS_IMAGE

# calculate image partition details for the boot and root partitions in
the Raspberry Pi OS image

###### ATTENTION: The output text of the 'Sector size' is locate
dependent and must be adjusted to your Country ! :-|

# sector_size=$(fdisk -l $RPOS_IMAGE | grep 'Sector size' | awk '{print
$4}')

sector_size=$(fdisk -l $RPOS_IMAGE | grep -m1 'Sektorgröße'| awk
'{print $3}')
start_sector_boot=$(fdisk -l $RPOS_IMAGE | grep -m1 "^${RPOS_IMAGE}1" |
awk '{print $2}')
```

```
start_sector_root=$(fdisk -l $RPOS_IMAGE | grep -m1 "^${RPOS_IMAGE}2" |
awk '{print $2}')
start_sector_bytes_boot=$((sector_size * start_sector_boot))
start_sector_bytes_root=$((sector_size * start_sector_root))

echo "Image sector size: $sector_size"
echo "Boot partition starts at sector $start_sector_boot (byte offset
$start_sector_bytes_boot)"
echo "Root partition starts at sector $start_sector_root (byte offset
$start_sector_bytes_root)"

# mount the first partition of the Raspberry Pi OS image (the boot
partition)
sudo mount -o loop,offset=$start_sector_bytes_boot $RPOS_IMAGE
$RPI_BOOT_MNT
read -p "boot mounted? " -n1 -s

if $RECREATED_RPOS_IMAGE;then
  ## Preconfigure username & password, enable SSH

  echo "${USERNAME}:${PASSWORD_HASH}"| sudo tee $RPI_BOOT_MNT/userconf
  sudo touch $RPI_BOOT_MNT/ssh # enable SSH
fi



## Build Linux Kernel
# see
https://www.raspberrypi.com/documentation/computers/linux_kernel.html#b
uilding-the-kernel
KERNEL_IMAGE=$WORK_DIR/linux/arch/arm64/boot/Image

REBUILT_KERNEL=false
if $REBUILD_KERNEL || $RECREATED_RPOS_IMAGE || ! [ -e $KERNEL_IMAGE
];then
  REBUILT_KERNEL=true
  # download linux kernel if necessary
  if ! [ -e $KERNEL_IMAGE ];then
    rm -r linux # remove old files
    git clone --depth=1 https://github.com/raspberrypi/linux
  fi

  cd linux || exit 1
  # create a .config file
  # this is the default config for Raspberry Pi 4
  KERNEL=kernel8
  make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
  # Use the kvm_guest config as the base defconfig, which is suitable
for qemu
  make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- kvm_guest.config

  ## manual kernel configuration DON'T DO THIS ON A MACHINE OF
```

```
DIFFERENT ARCHITECTURE
  # make menu config
  sed -i 's/# CONFIG_DRM_QXL is not set/CONFIG_DRM_QXL=m/' .config
  sed -i 's/# CONFIG_FB_SIMPLE is not set/CONFIG_FB_SIMPLE=y/' .config
  sed -i 's/# CONFIG_FB_VIRTUAL is not set/CONFIG_FB_VIRTUAL=y/'
.config
  sed -i 's/# CONFIG_NETLINK_DIAG is not set/CONFIG_NETLINK_DIAG=y/'
.config

  # Build the kernel
  make -j12 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules
dtbs
  ## Install kernel modules into the guest filesystem
  # mount the second partition of the Raspberry Pi OS image (the root
partition)
else
  cd linux || exit 1
fi
# sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
# sudo cp arch/arm64/boot/Image mnt/boot/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb $RPI_BOOT_MNT
sudo cp arch/arm64/boot/dts/overlays/*.dtb* $RPI_BOOT_MNT/overlays/
sudo cp arch/arm64/boot/dts/overlays/README $RPI_BOOT_MNT/overlays/
read -p "Press key to continue.. " -n1 -s
sudo umount $RPI_BOOT_MNT
sudo mount -o loop,offset=$start_sector_bytes_root $RPOS_IMAGE
$RPI_ROOT_MNT
read -p "root mounted? " -n1 -s
# Install kernel modules into the guest filesystem
sudo env PATH=$PATH make -j12 ARCH=arm64 CROSS_COMPILE=aarch64-linux-
gnu- INSTALL_MOD_PATH=$RPI_ROOT_MNT modules_install


sudo umount $RPI_ROOT_MNT

cd .. || exit 1 # return to parent directory


sudo chown libvirt-qemu:kvm $RPOS_IMAGE



## Run unregistered VM directly using qemu (CLI only)
# qemu-system-aarch64 -machine virt -cpu cortex-a72 -smp 6 -m 4G \
#     -kernel $KERNEL_IMAGE -append "root=/dev/vda2 rootfstype=ext4 rw
panic=0 console=ttyAMA0" \
#     -drive format=raw,file=$RPOS_IMAGE,if=none,id=hd0,cache=writeback
\
#     -device virtio-blk,drive=hd0,bootindex=0 \
#     -netdev user,id=mynet,hostfwd=tcp::2222-:22 \
#     -device virtio-net-pci,netdev=mynet \
```

```
#     -monitor telnet:127.0.0.1:5555,server,nowait

## Create registered VM for virt-manager (with graphics!)
echo "
<domain type='qemu'>
  <name>$VM_NAME</name>
  <memory unit='GiB'>4</memory>
  <vcpu placement='static'>6</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>$KERNEL_IMAGE</kernel>
    <cmdline>root=/dev/vda2 rootfstype=ext4 rw panic=0
console=ttyAMA0</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model>cortex-a72</model>
  </cpu>
  <devices>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='$RPOS_IMAGE'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
      <model type='virtio'/>
    </interface>
    <!--<filesystem type='mount' accessmode='mapped'>
      <source dir='$QBM_SOURCE'/>
      <target dir='$SHARED_FS_TAG'/>
    </filesystem>-->
    <serial type='pty'>
      <source path='/dev/pts/4'/>
      <target type='system-serial' port='0'>
        <model name='pl011'/>
      </target>
      <alias name='serial0'/>
    </serial>
    <controller type='usb' index='0' model='qemu-xhci' ports='15'>
      <address type='pci' domain='0x0000' bus='0x07' slot='0x00'
function='0x0'/>
    </controller>
    <input type='mouse' bus='usb'/>
    <input type='keyboard' bus='usb'/>
    <graphics type='spice' autoport='yes'>
      <listen type='address'/>
      <image compression='off'/>
      <gl enable='no'/>
    </graphics>
    <audio id='1' type='none'/>
    <video>
```

```
      <model type='virtio' heads='1' primary='yes'/>
    </video>
  </devices>
</domain>

" > raspi_vm.config
virsh define raspi_vm.config
virsh start "$VM_NAME"



BLACK='\033[0;30m'
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[0;33m'
BLUE='\033[0;34m'
MAGENTA='\033[0;35m'
CYAN='\033[0;36m'
WHITE='\033[0;37m'
NC='\033[0m' # No Color
echo -e "$YELLOW
It is normal for the VM window to display while booting:
'Guest has not initialized the display (yet).'

Wait for a minute, after which the console log-in should display.
In the virt-manager VM window, under the menu _View > Consoles > Serial
1_ you can switch to the text console while the graphics aren't running
yet.

$NC"
```

Wegwerf- Raspi in Docker

Das ist ja dann wieder einfach ein

```
  docker run -it -p 5022:5022 ghcr.io/carlosperate/qemu-rpi-os-
lite:bullseye-latest
```

und schon hat man einen Wegwerf- Raspi auf localhost:5022. Leider ist das Image schon älter.

Das mit dem älteren Image wurde zum Problem, als die Test-Installation Zuul-AC unter bullseye und Python 3.9 immer in eine Async-Task-Error- Katastrophe abrauschte  Also mußte nun doch ein neueres Image her.

Also wieder zurück auf https://github.com/stko/rpi-os-custom-image und das Repository geclont.

Dann erstmal in `download_os.py` die Parameter auf einen aktuellen Raspi- Kernel ergänzen,

Wenn das dann durch ist und man neue Images hat, kann man dann mit

```
docker run -it --rm --name rpi-os-d4eb3d25 -p 5022:5022 -v
/home/steffen/Playground/rpi-os-custom-image/rpiosimage/2025-10-01-raspios-
trixie-armhf-lite-autologin-ssh-expanded.img:/sdcard/filesystem.img
lukechilds/dockerpi:vm
```

auch neuere Raspians starten (darauf achten, dass man das Richtige mit ssh und expanded erwischt), zumindest so lange der von Qemu verwendete alte und hardvercodete Custom- Kernel das neuere Image noch irgendwie ans Laufen bekommt. Irgendwann ist es also wieder vorbei..

Die Änderungen, die man im Raspi macht, bleiben im Image gespeichert - Man sollte sich also, nachdem man das erzeugte Image das erste Mal gestartet hat und dies dann all seine Aktualisierungs-Zyklen durchlaufen hat, dies dann beenden und als „Original“ an die Seite legen und nur mit einer Kopie arbeiten, denn sonst verdirbt man sich den jungfräulichen Charakter des Images

Startet man den Docker ohne Parameter, wird als Umgebung ein Raspi 1 angenommen mit nur 256MB Speicher, was beim Ansible- Lauf dann zum Crash wegen Speichermangel führt. Gibt man als letzten Parameter im Befehl ein `pi3` an, hängt das Ding beim Booten in irgendeiner (virtuellen) usb- init- Problematik, aber mit `pi2` geht's...

Damit kann mit Ansible komplett automatisch seine Anwendungen installieren und dann ausprobieren. Beim weiteren Rumspielen merkt man dann:

- der Docker- Container wiederum ist ja ein Qemu- Emulator, der den Raspi emuliert
- die Ports, die der Quemu in den Docker durchleitet, sind ja hardcoded im Image und hier nur der Port 5022 für SHH
- d.h. egal mit welchen zusätzlichen Ports man den Container auch startet, vom Raspi über den Qemu kommt nur `ssh`

Nur wie kommt man jetzt an die Ports der Applikation, die man doch eigentlich testen wil??

Nach etwas Kopfkratzen dann die Lösung: ssh- port- forwarding.. (Parameter -L...)

Damit kann einen Port auf SSH- Client- Seite mit einem vom SSH- Server sichtbaren Ports eines Zielsystems verbinden, man kann also quasi über Bande spielen.

Mit dem Kommando

```
ssh -p 5022 -L8080:127.0.0.1:8000 pi@localhost
```

verbindet man Port 8000 des Raspis mit dem lokalen Port 8000 und kommt per `http://localhost:8080/` nun **endlich** auf die Anwendung, die es zu testen gilt.

[1)]

kopiert und zweckoptimiert von
https://gist.github.com/emendir/922d6914a1705ed2e8e4e96db726c422