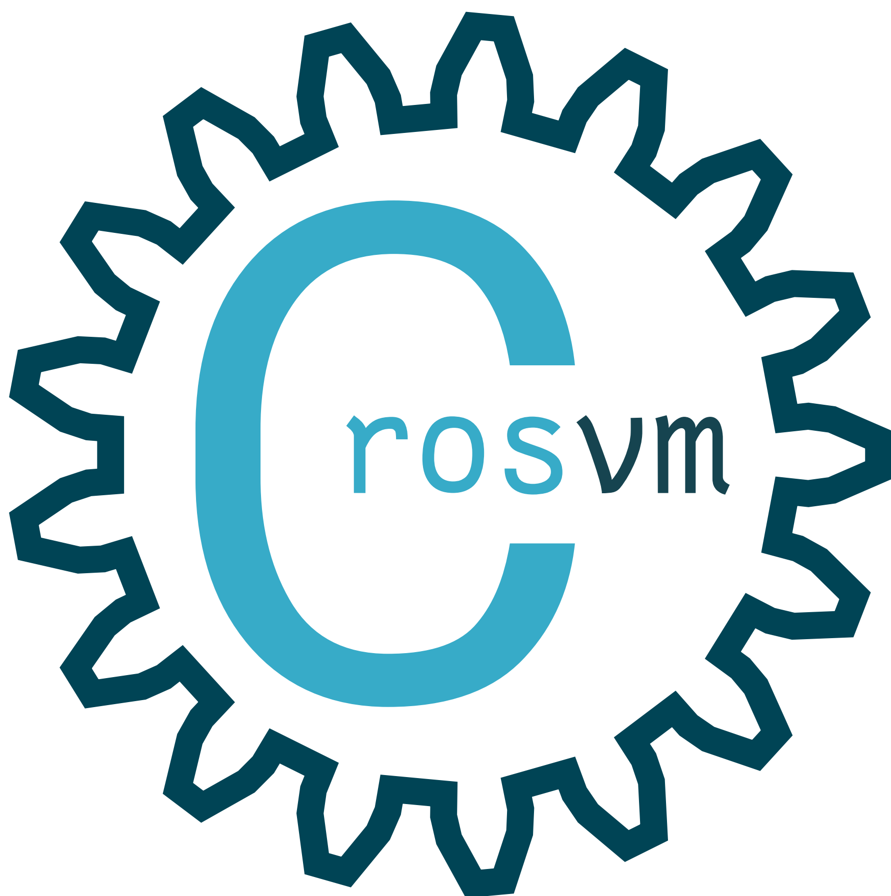
Introduction

The crosvm project is a hosted (a.k.a. type-2) virtual machine monitor.

crosvm runs untrusted operating systems along with virtualized devices. Initially intended to be used with KVM and Linux, crosvm supports multiple kinds of hypervisors. crosvm is focussed on safety within the programming language and a sandbox around the virtual devices to protect the host from attack in case of exploits in crosvm itself.

Other programs similar to crosvm are QEMU and VirtualBox. An operating system, made of a root file system image and a kernel binary, are given as input to crosvm and then crosvm will run the operating system using the platform's hypervisor.

- Announcements
- Developer Mailing List
- #crosvm on matrix.org
- Source code
    - GitHub mirror
    - API documentation, useful for searching API.
    - Files for this book are under /docs/.
- Public issue tracker
    - For Googlers: See go/crosvm#filing-bugs.



Building Crosvm

This page describes how to build and develop crosvm on linux. If you are targeting another OS such as ChromeOS, please see Integration

Checking out

Obtain the source code via git clone.

```
git clone https://chromium.googlesource.com/crosvm/crosvm
```

Setting up the development environment

Crosvm uses submodules to manage external dependencies. Initialize them via:

```
git submodule update --init
```

It is recommended to enable automatic recursive operations to keep the submodules in sync with the main repository (But do not push them, as that can conflict with `repo`):

```
git config submodule.recurse true
git config push.recurseSubmodules no
```

Crosvm development best works on Debian derivatives. We provide a script to install the necessary packages on Debian, Ubuntu or gLinux:

```
./tools/install-deps
```

For other systems, please see below for instructions on Using the development container.

Using the development container

We provide a Debian container with the required packages installed. With Docker installed, it can be started with:

```
./tools/dev_container
```

The container image is big and may take a while to download when first used. **Once started, you can follow all instructions in this document within the container shell.**

Instead of using the interactive shell, commands to execute can be provided directly:

```
./tools/dev_container cargo build
```

Note: The container and build artifacts are preserved between calls to `./tools/dev_container`. If you wish to start fresh, use the `--reset` flag.

Building a binary

If you simply want to try crosvm, run `cargo build`. Then the binary is generated at `./target/debug/crosvm`. Now you can move to Example Usage.

If you want to enable additional features, use the `--features` flag. (e.g. `cargo build --features=gdb`)

Development

Iterative development

You can use cargo as usual for crosvm development to `cargo build` and `cargo test` single crates that you are working on.

If you are working on aarch64 specific code, you can use the `test_target` tool to instruct cargo to build for aarch64 and run tests on a VM:

```
./tools/test_target set vm:aarch64 && source .envrc
cd mycrate && cargo test
```

The script will start a VM for testing and write environment variables for cargo to `.envrc`. With those `cargo build` will build for aarch64 and `cargo test` will run tests inside the VM.

The aarch64 VM can be managed with the `./tools/aarch64vm` script.

Note: See Cross-compilation for notes on cross-compilation.

Running all tests

Crosvm cannot use `cargo test --workspace` because of various restrictions of cargo. So we have our own test runner:

```
./tools/run_tests
```

Which will run all tests locally. Since we have some architecture-dependent code, we also have the option of running tests within an aarch64 VM:

```
./tools/run_tests --target=vm:aarch64
```

When working on a machine that does not support cross-compilation (e.g. gLinux), you can use the dev container to build and run the tests.

```
./tools/dev_container ./tools/run_tests --target=vm:aarch64
```

It is also possible to run tests on a remote machine via ssh. The target architecture is automatically detected:

```
./tools/run_tests --target=ssh:hostname
```

However, it is your responsibility to make sure the required libraries for crosvm are installed and password-less authentication is set up. See `./tools/impl/testvm/cloud_init.yaml` for hints on what the VM has installed.

Presubmit checks

To verify changes before submitting, use the `presubmit` script:

```
./tools/presubmit
```

This will run clippy, formatters and runs all tests. The presubmits will use the dev container to build for other platforms if your host is not set up to do so.

To run checks faster, they can be run in parallel in multiple tmux panes:

```
./tools/presubmit --tmux
```

The `--quick` variant will skip some slower checks, like building for other platforms altogether:

```
./tools/presubmit --quick
```

Cross-compilation

Crosvm is built and tested on x86, aarch64 and armhf. Your system needs some setup work to be able to cross-comple for other architectures, hence it is recommended to use the development container, which will have everything configured.

Note: Cross-compilation is **not supported on gLinux**. Please use the development container.

Enable foreign architectures

Your host needs to be set up to allow installation of foreign architecture packages.

On Debian this is as easy as:

```
sudo dpkg --add-architecture arm64
sudo dpkg --add-architecture armhf
sudo apt update
```

On ubuntu this is a little harder and needs some manual modifications of APT sources.

With that enabled, the following scripts will install the needed packages:

```
./tools/install-aarch64-deps
./tools/install-armhf-deps
```

Configuring wine and mingw64

Crosvm is also compiled and tested on windows. Some limited testing can be done with mingw64 and wine on linux machines. Use the provided setup script to install the needed dependencies.

```
./tools/install-mingw64-deps
```

Configure cargo for cross-compilation

Cargo requries additional configuration to support cross-compilation. You can copy the provided example config to your cargo configuration:

```
cat .cargo/config.debian.toml >> ${CARGO_HOME:-~/.cargo}/config.toml
```

Known issues

- Devices can't be jailed if `/var/empty` doesn't exist. `sudo mkdir -p /var/empty` to work around this for now.
- You need read/write permissions for `/dev/kvm` to run tests or other crosvm instances. Usually it's owned by the `kvm` group, so `sudo usermod -a -G kvm $USER` and then log out and back in again to fix this.
- Some other features (networking) require `CAP_NET_ADMIN` so those usually need to be run as root.

Building Crosvm on Windows

This page describes how to build and develop crosvm on windows. If you are targeting linux, please see Building Crosvm on linux

NOTE: Following instruction assume that

- git is installed and `git` command exists in your `Env:PATH`
- the commands are run in powershell

Create base directory - `C:\src`

```
mkdir C:\src
cd C:\src
```

Checking out

Obtain the source code via git clone.

```
git clone https://chromium.googlesource.com/crosvm/crosvm
```

Setting up the development environment

Crosvm uses submodules to manage external dependencies. Initialize them via:

```
cd crosvm
git submodule update --init
```

It is recommended to enable automatic recursive operations to keep the submodules in sync with the main repository (But do not push them, as that can conflict with `repo`):

```
git config submodule.recurse true
git config push.recurseSubmodules no
```

`install-deps.ps1` install the necessary tools needed to build crosvm on windows. In addition to installing the scripts, the script also sets up environment variables.

The below script may prompt you to install msvc toolchain via Visual Studio community edition.

```
Set-ExecutionPolicy Unrestricted -Scope CurrentUser
./tools/install-deps.ps1
```

NOTE: Above step sets up enviroment variables. You may need to either start a new powershell session or reload the environemnt variables,

Build crosvm

```
cargo build --features all-msvc64,whpx
```

Testing

Crosvm runs on a variety of platforms with a significant amount of platform-specific code. Testing on all the supported platforms is crucial to keep crosvm healthy.

Unit Tests

Unit tests are your standard rust tests embedded with the rest of the code in `src/` and wrapped in a `#[cfg(test)]` attribute.

Unit tests **cannot make any guarantees on the runtime environment**. Avoid doing the following in unit tests:

- Avoid kernel features such as io_uring or userfaultfd, which may not be available on all kernels.
- Avoid functionality that requires privileges (e.g. CAP_NET_ADMIN)
- Avoid spawning threads or processes
- Avoid accessing kernel devices
- Avoid global state in unit tests

This allows us to execute unit tests for any platform using emulators such as qemu-static or wine64. It also allows them to execute quickly with parallel execution.

Integration tests

Cargo has native support for integration testing. Integration tests are written just like unit tests, but live in a separate directory at `tests/`.

Integration tests **guarantee that the test has privileged access to the test environment** and that tests are executed exclusively on a system to prevent conflicts with each other.

This allows tests to do all the things unit tests cannot do, at the cost of slower execution.

End To End (E2E) tests

End to end tests live in the `e2e_tests` crate. The crate provides a framework to boot a guest with crosvm and execut commands in the guest to validate functionality at a high level.

E2E tests are executed just like integration tests.

Downstream Product tests

Each downstream product that uses crosvm is performing their own testing, e.g. ChromeOS is running high level testing of its VM features on ChromeOS hardware, while AOSP is running testing

of their VM features on AOSP hardware.

Upstream crosvm is not involved in these tests and they are not executed in crosvm CI.

Platorms tested

The platforms below can all be tested using `tools/run_tests -p $platform`. The table indicates how these tests are executed:

Platform	Build	Unit Tests	Integration Tests	E2E Tests
x86_64 (linux)	✅	✅	✅	✅
aarch64 (linux)	✅	✅ (qemu-static[1])	✅ (qemu[2])	❌
armhf (linux)	✅	✅ (qemu-static[1])	❌	❌
mingw64[3] (linux)	🚧	🚧 (wine64)	❌	❌
mingw64[3] (windows)	🚧	🚧	🚧	❌

Crosvm CI will use the same configuration as `tools/run_tests`.

[1] qemu-static-aarch64 or qemu-static-arm translate instructions into x86 and executes them on the host kernel. This works well for unit tests, but will fail when interacting with platform specific kernel features.

[2] run_tests will launch a VM for testing in the background. This VM is using full system emulation, which causes tests to be slow. Also not all aarch64 features are properly emulated, which prevents us from running e2e tests.

[3] Windows builds of crosvm are a work in progress. Some tests are executed via wine64 on linux

Running Crosvm

This chapter includes instructions on how to run crosvm.

- Example Usage: Functioning examples to get started.
- Advanced Usage: Details on how to enable and configure features and devices of crosvm.
- Custom Kernel / Rootfs: Instructions on how to build a kernel and rootfs for crosvm.
- Options and Configuration Files: How to specify command-line options and use configuration files
- System Requirements: Host and guest requirements for running crosvm
- Features: Feature flags available when building crosvm

Example Usage

This section will explain how to use a prebuilt Ubuntu image as the guest OS. If you want to prepare a kernel and rootfs by yourself, please see Building crosvm.

The example code for this guide is available in tools/examples

Run a simple Guest OS (using virt-builder)

To run a VM with crosvm, we need two things: A kernel binary and a rootfs. You can build those yourself or use prebuilt cloud/vm images that some linux distributions provide.

Preparing the guest OS image

One of the more convenient ways to customize these VM images is to use virt-builder from the `libguestfs-tools` package.

```
# Build a simple ubuntu image and create a user with no password.
virt-builder ubuntu-20.04 \
    --run-command "useradd -m -g sudo -p '' $USER ; chage -d 0 $USER" \
    -o ./rootfs
```

Extract the Kernel (And initrd)

Crosvm directly runs the kernel instead of using the bootloader. So we need to extract the kernel binary from the image. virt-builder has a tool for that:

```
virt-builder --get-kernel ./rootfs -o .
```

The kernel binary is going to be saved in the same directory.

Note: Most distributions use an init ramdisk, which is extracted at the same time and needs to be passed to crosvm as well.

Launch the VM

With all the files in place, crosvm can be run:

```
# Run crosvm without sandboxing.
# The rootfs is an image of a partitioned hard drive, so we need to tell
# the kernel which partition to use (vda5 in case of ubuntu-20.04).
cargo run --no-default-features -- run \
    --disable-sandbox \
    --rwdisk ./rootfs \
    --initrd ./initrd.img-* \
    -p "root=/dev/vda5" \
    ./vmlinuz-*
```

The full source for this example can be executed directly:

```
./tools/examples/example_simple
```

Add Networking Support

Networking support is easiest set up with a TAP device on the host, which can be done with:

```
./tools/examples/setup_network
```

The script will create a TAP device called `crosvm_tap` and sets up routing. For details, see the instructions for network devices.

With the `crosvm_tap` in place we can use it when running crosvm:

```
# Use the previously configured crosvm_tap device for networking.
cargo run -- run \
    --disable-sandbox \
    --rwdisk ./rootfs \
    --initrd ./initrd.img-* \
    --tap-name crosvm_tap \
    -p "root=/dev/vda5" \
    ./vmlinuz-*
```

To use the network device in the guest, we need to assign it a static IP address. In our example guest this can be done via a netplan config:

```
# Configure network with static IP 192.168.10.2

network:
    version: 2
    renderer: networkd
    ethernets:
        enp0s4:
            addresses: [192.168.10.2/24]
            nameservers:
                addresses: [8.8.8.8]
            gateway4: 192.168.10.1
```

Which can be installed when building the VM image:

```
builder_args=(
    # Create user with no password.
    --run-command "useradd -m -g sudo -p '' $USER ; chage -d 0 $USER"

    # Configure network via netplan config in 01-netcfg.yaml
    --hostname crosvm-test
    --copy-in "$SRC/guest/01-netcfg.yaml:/etc/netplan/"

    # Install sshd and authorized key for the user.
    --install openssh-server
    --ssh-inject "$USER:file:$HOME/.ssh/id_rsa.pub"

    -o rootfs
)
virt-builder ubuntu-20.04 "${builder_args[@]}"
```

This also allows us to use SSH to access the VM. The script above will install your `~/.ssh/id_rsa.pub` into the VM, so you'll be able to SSH from the host to the guest with no password:

```
ssh 192.168.10.2
```

The full source for this example can be executed directly:

```
./tools/examples/example_network
```

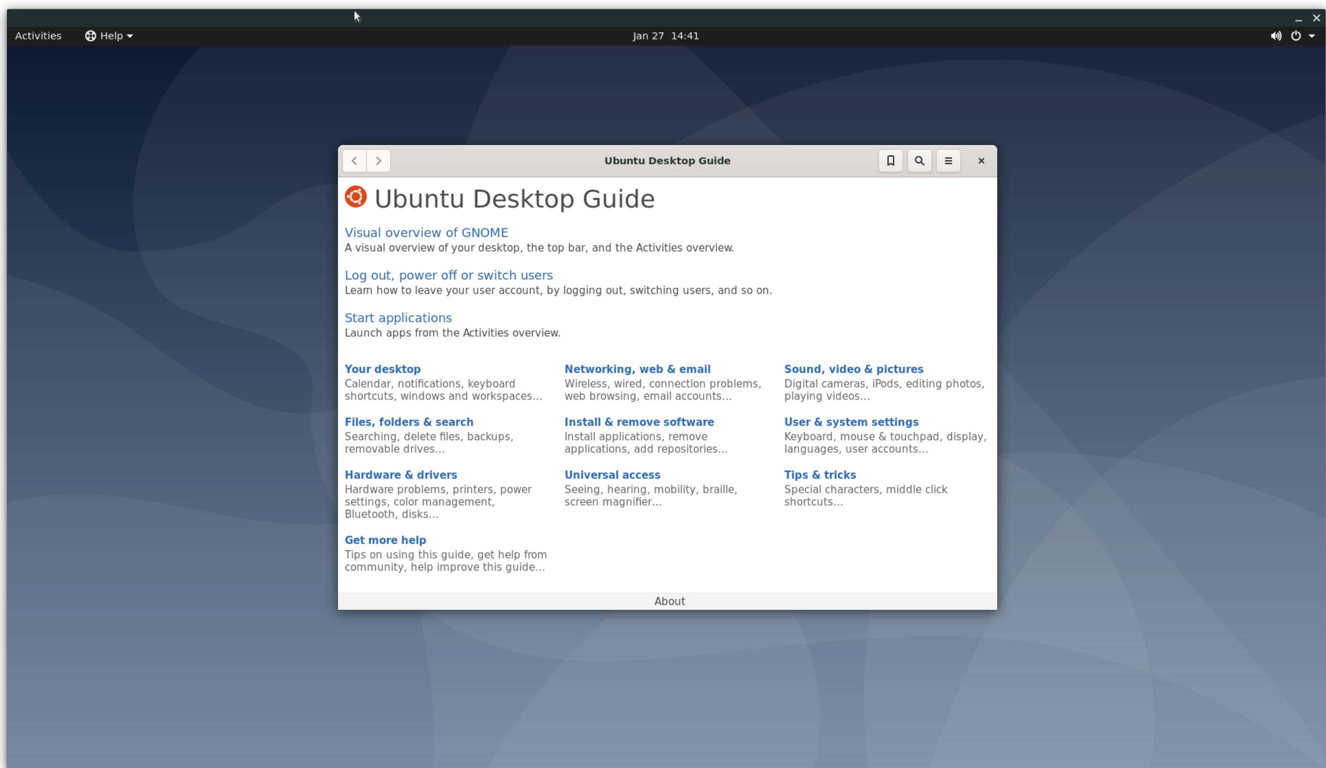
Add GUI support

First you'll want to add some desktop environment to the VM image:

```
builder_args=(
    # Create user with no password.
    --run-command "useradd -m -g sudo -p '' $USER ; chage -d 0 $USER"

    # Configure network. See ./example_network
    --hostname crosvm-test
    --copy-in "$SRC/guest/01-netcfg.yaml:/etc/netplan/"

    # Install a desktop environment to launch
    --install xfce4

    -o rootfs
)
virt-builder ubuntu-20.04 "${builder_args[@]}"
```

Then you can use the `--gpu` argument to specify how gpu output of the VM should be handled. In this example we are using the virglrenderer backend and output into an X11 window on the host.

```
# Enable the GPU and keyboard/mouse input. Since this will be a much heavier
# system to run we also need to increase the cpu/memory given to the VM.
# Note: GDM does not allow you to set your password on first login, you have to
#       log in on the command line first to set a password.
cargo run --features=gpu,x,virgl_renderer -- run \
    --cpus 4 \
    --mem 4096 \
    --disable-sandbox \
    --gpu backend=virglrenderer,width=1920,height=1080 \
    --display-window-keyboard \
    --display-window-mouse \
    --tap-name crosvm_tap \
    --rwdisk ./rootfs \
    --initrd ./initrd.img-* \
    -p "root=/dev/vda5" \
    ./vmlinuz-*
```



The full source for this example can be executed directly (Note, you may want to run setup_networking first):

```
./tools/examples/example_desktop
```

Advanced Usage

To see the usage information for your version of crosvm, run `crosvm` or `crosvm run --help`.

Boot a Kernel

To run a very basic VM with just a kernel and default devices:

```
crosvm run "${KERNEL_PATH}"
```

The compressed kernel image, also known as bzImage, can be found in your kernel build directory in the case of x86 at `arch/x86/boot/bzImage`.

Rootfs

With a disk image

In most cases, you will want to give the VM a virtual block device to use as a root file system:

```
crosvm run -b "${ROOT_IMAGE},root,ro" "${KERNEL_PATH}"
```

The root image must be a path to a disk image formatted in a way that the kernel can read. Typically this is a squashfs image made with `mksquashfs` or an ext4 image made with `mkfs.ext4`. By specifying the `root` flag, the kernel is automatically told to use that image as the root, and therefore it can only be given once. The `ro` flag also makes the disk image read-only for the guest. More disks images can be given with `-b` or `--block` if needed.

To run crosvm with a writable rootfs, just remove the `ro` flag from the command-line above.

> **WARNING:** Writable disks are at risk of corruption by a malicious or malfunctioning guest OS.

Without the `root` flag, mounting a disk image as the root filesystem requires to pass the corresponding kernel argument manually using the `-p` option:

```
crosvm run --block "${ROOT_IMAGE}" -p "root=/dev/vda" bzImage
```

> **NOTE:** If more disks arguments are added prior to the desired rootfs image, the

> `root=/dev/vda` must be adjusted to the appropriate letter.

With virtiofs

Linux kernel 5.4+ is required for using virtiofs. This is convenient for testing. The file system must be named "mtd*" or "ubi*".

```
crosvm run --shared-dir "/:mtdfake:type=fs:cache=always" \
    -p "rootfstype=virtiofs root=mtdfake" bzImage
```

Device emulation

Crosvm supports several emulated devices and 15+ types of virtio devices. See "Device" chapter for the details.

Control Socket

If the control socket was enabled with `-s`, the main process can be controlled while crosvm is running. To tell crosvm to stop and exit, for example:

> **NOTE:** If the socket path given is for a directory, a socket name underneath that path will be generated based on crosvm's PID.

```
crosvm run -s /run/crosvm.sock ${USUAL_CROSVM_ARGS}
    <in another shell>
crosvm stop /run/crosvm.sock
```

> **WARNING:** The guest OS will not be notified or gracefully shutdown.

This will cause the original crosvm process to exit in an orderly fashion, allowing it to clean up any OS resources that might have stuck around if crosvm were terminated early.

Multiprocess Mode

By default crosvm runs in multiprocess mode. Each device that supports running inside of a sandbox will run in a jailed child process of crosvm. The sandbox can be disabled for testing with the `--disable-sandbox` option.

GDB Support

crosvm supports GDB Remote Serial Protocol to allow developers to debug guest kernel via GDB (**x86_64 or AArch64 only**).

You can enable the feature by `--gdb` flag:

```
# Use uncompressed vmlinux
crosvm run --gdb <port> ${USUAL_CROSVM_ARGS} vmlinux
```

Then, you can start GDB in another shell.

```
gdb vmlinux
(gdb) target remote :<port>
(gdb) hbreak start_kernel
(gdb) c
<start booting in the other shell>
```

For general techniques for debugging the Linux kernel via GDB, see this kernel documentation.

Defaults

The following are crosvm's default arguments and how to override them.

- 256MB of memory (set with `-m`)
- 1 virtual CPU (set with `-c`)
- no block devices (set with `-b`, `--block`)
- no network device (set with `--net`)
- only the kernel arguments necessary to run with the supported devices (add more with `-p`)
- run in multiprocess mode (run in single process mode with `--disable-sandbox`)
- no control socket (set with `-s`)

Exit code

Crosvm will exit with a non-zero exit code on failure.

See CommandStatus for meaning of the major exit codes.

Custom Kernel / Rootfs

This document explains how to build a custom kernel and use debootstrab to build a rootfs for running crosvm.

For an easier way to get started with prebuilt images, see Example Usage

Build a kernel

The linux kernel in chromiumos comes preconfigured for running in a crosvm guest and is the easiest to build. You can use any mainline kernel though as long as it's configured for para-virtualized (virtio) devices

If you are using the chroot for ChromiumOS development, you already have the kernel source. Otherwise, you can clone it:

```
git clone --depth 1 -b chromeos-5.10
https://chromium.googlesource.com/chromiumos/third_party/kernel
```

Either way that you get the kernel, the next steps are to configure and build the bzImage:

```
CHROMEOS_KERNEL_FAMILY=termina ./chromeos/scripts/prepareconfig container-vm-x86_64
make olddefconfig
make -j$(nproc) bzImage
```

This kernel does not build any modules, nor does it support loading them, so there is no need to worry about an initramfs, although they are supported in crosvm.

Build a rootfs disk

This stage enjoys the most flexibility. There aren't any special requirements for a rootfs in crosvm, but you will at a minimum need an init binary. This could even be `/bin/bash` if that is enough for your purposes. To get you started, a Debian rootfs can be created with debootstrap. Make sure to define `$CHROOT_PATH`.

```
truncate -s 20G debian.ext4
mkfs.ext4 debian.ext4
mkdir -p "${CHROOT_PATH}"
sudo mount debian.ext4 "${CHROOT_PATH}"
sudo debootstrap stable "${CHROOT_PATH}" http://deb.debian.org/debian/
sudo chroot "${CHROOT_PATH}"
passwd
echo "tmpfs /tmp tmpfs defaults 0 0" >> /etc/fstab
echo "tmpfs /var/log tmpfs defaults 0 0" >> /etc/fstab
echo "tmpfs /root tmpfs defaults 0 0" >> /etc/fstab
echo "sysfs /sys sysfs defaults 0 0" >> /etc/fstab
echo "proc /proc proc defaults 0 0" >> /etc/fstab
exit
sudo umount "${CHROOT_PATH}"
```

> Note: If you run crosvm on a testing device (e.g. Chromebook in Developer mode), another option is to share the host's rootfs with the guest via virtiofs. See the virtiofs usage.

You can simply create a disk image as follows:

```
fallocate --length 4G disk.img
mkfs.ext4 ./disk.img
```

Command line options and configuration files

It is possible to configure a VM through command-line options and/or a JSON configuration file.

The names and format of configurations options are consistent between both ways of specifying, however the command-line includes options that are deprecated or unstable, whereas the configuration file only allows stable options. This section reviews how to use both.

Command-line options

Command-line options generally take a set of key-value pairs separated by the comma (`,`) character. The acceptable key-values for each option can be obtained by passing the `--help` option to a crosvm command:

```
crosvm run --help
...
  -b, --block       parameters for setting up a block device.
                    Valid keys:
                        path=PATH - Path to the disk image. Can be specified
                            without the key as the first argument.
                        ro=BOOL - Whether the block should be read-only.
                            (default: false)
                        root=BOOL - Whether the block device should be mounted
                            as the root filesystem. This will add the required
                            parameters to the kernel command-line. Can only be
                            specified once. (default: false)
                        sparse=BOOL - Indicates whether the disk should support
                            the discard operation. (default: true)
                        block-size=BYTES - Set the reported block size of the
                            disk. (default: 512)
                        id=STRING - Set the block device identifier to an ASCII
                            string, up to 20 characters. (default: no ID)
                        direct=BOOL - Use O_DIRECT mode to bypass page cache.
                            (default: false)
...
```

From this help message, we see that the `--block` or `-b` option accepts the `path` , `ro` , `root` , `sparse` , `block-size` , `id` , and `direct` keys. Keys which default value is mentioned are optional, which means only the `path` key must always be specified.

One example invocation of the `--block` option could be:

```
--block path=/path/to/bzImage,root=true,block-size=4096
```

Keys taking a boolean parameters can be enabled by specifying their name witout any value, so the previous option can also be written as

```
--block path=/path/to/bzImage,root,block-size=4096
```

Also, the name of the first key can be entirely omitted, which further simplifies our option as:

```
--block /path/to/bzImage,root,block-size=4096
```

Configuration files

Configuration files are specified using the `--cfg` argument. Here is an example configuration file specifying a basic VM with a few devices:

```
{
    "kernel": "/path/to/bzImage",
    "cpus": {
        "num-cores": 8
    },
    "mem": {
        "size": 2048
    },
    "block": [
        {
            "path": "/path/to/root.img",
            "root": true
        }
    ],
    "serial": [
        {
            "type": "stdout",
            "hardware": "virtio-console",
            "console": true,
            "stdin": true
        }
    ],
    "net": [
        {
            "tap-name": "crosvm_tap"
        }
    ]
}
```

The equivalent command-line options corresponding to this configuration file would be:

```
--kernel path/to/bzImage \
--cpus num-cores=8 --mem size=2048 \
--block path=/path/to/root.img,root \
--serial type=stdout,hardware=virtio-console,console,stdin \
--net tap-name=crosvm_tap
```

Or, if we apply the simplification rules discussed in the previous section:

```
--kernel /path/to/bzImage \
--cpus 8 --mem 2048 \
--block /path/to/root.img,root \
--serial stdout,hardware=virtio-console,console,stdin \
--net tap-name=crosvm_tap
```

Combining configuration files and command-line options

One useful use of configuration files is to specify a base configuration that can be augmented or modified.

Configuration files and other command-line options can be specified together. When this happens, the command-line parameters will be merged into the initial configuration created by the configuration file, regardless of their position relative to the `--cfg` argument and even if they come before it.

The effect of command-line arguments redefining items of the configuration file depends on the nature of said items. If an item can be specified several times (like a block device), then the command-line arguments will augment the configuration file. For instance, considering this configuration file `vm.json`:

```
{
    "kernel": "/path/to/bzImage",
    "block": [
        {
            "path": "/path/to/root.img",
            "root": true
        }
    ]
}
```

And the following crosvm invocation:

```
crosvm run --cfg vm.json --block /path/to/home.img
```

Then the created VM will have two block devices, the first one pointing to `root.img` and the second one to `home.img`.

For options that can be specified only once, like `--kernel`, the one specified on the command-line will take precedence over the one in the configuration file. For instance, with the same `vm.json` file and the following command-line:

```
crosvm run --cfg vm.json --kernel /path/to/another/bzImage
```

Then the loaded kernel will be `/path/to/another/bzImage`, and the `kernel` option in the configuration file will become a no-op.

System Requirements

A Linux kernel with KVM support (check for `/dev/kvm`) is required to run crosvm. In order to run certain devices, there are additional system requirements:

- `virtio-wayland` - The `memfd_create` syscall, introduced in Linux 3.17, and a Wayland compositor.
- `vsock` - Host Linux kernel with vhost-vsock support, introduced in Linux 4.8.
- `multiprocess` - Host Linux kernel with seccomp-bpf and Linux namespacing support.
- `virtio-net` - Host Linux kernel with TUN/TAP support (check for `/dev/net/tun`) and running with `CAP_NET_ADMIN` privileges.

Features

Feature flags of the `crosvm` crate control which features are included in the binary. These features can be enabled using Cargo's `--features` flag. Some features are enabled by default unless the Cargo `--no-default-features` flag is specified. See the `crosvm` crate documentation for details.

Programmatic Interaction Using the `crosvm_control` Library

Usage

`crosvm_control` provides a programmatic way to interface with crosvm as a substitute to the CLI.

The library itself is written in Rust, but a C/C++ compatible header (`crosvm_control.h`) is generated during the crosvm build and emitted to the Rust `OUT_DIR`. (See the `build.rs` script for more information).

The best practice for using `crosvm_control` from your project is to exclusively use the `crosvm_control.h` generated by the crosvm build. This ensures that there will never be a runtime version mismatch between your project and crosvm. Additionally, this will allow for build-time checks against the crosvm API.

During your project's build step, when building the crosvm dependency, the emitted `crosvm_control.h` should be installed to your project's include dir - overwriting the old version if present.

Changes

As `crosvm_control` is a externally facing interface to crosvm, great care must be taken when updating the API surface. Any breaking change to a `crosvm_control` entrypoint must be handled the same way as a breaking change to the crosvm CLI.

As a general rule, additive changes (such as adding new fields to the end of a struct, or adding a new API) are fine and should be integrated correctly with downstream projects so long as those projects follow the usage best practices. Changes that change the signature of any existing `crosvm_control` function will cause problems downstream and should be considered a breaking change.

(ChromeOS Developers Only)

For ChromeOS, it is possible to integrate a breaking change from upstream crosvm, but it should be avoided if at all possible. See here for more information.

Devices

This chapter describes emulated devices in crosvm. These devices work like hardware for the guest.

List of devices

Here is a (non-comprehensive) list of emulated devices provided by crosvm.

Emulated Devices

- `CMOS/RTC` - Used to get the current calendar time.
- `i8042` - Used by the guest kernel to exit crosvm.
- `serial` - x86 I/O port driven serial devices that print to stdout and take input from stdin.

VirtIO Devices

- `balloon` - Allows the host to reclaim the guest's memories.
- `block` - Basic read/write block device.
- `console` - Input and outputs on console.
- `fs` - Shares file systems over the FUSE protocol.
- `gpu` - Graphics adapter.
- `input` - Creates virtual human interface devices such as keyboards.
- `iommu` - Emulates an IOMMU device to manage DMA from endpoints in the guest.
- `net` - Device to interface the host and guest networks.
- `p9` - Shares file systems over the 9P protocol.
- `pmem` - Persistent memory.
- `rng` - Entropy source used to seed guest OS's entropy pool.
- `snd` - Encodes and decodes audio streams.
- `tpm` - Creates a TPM (Trusted Platform Module) device backed by libtpm2 simulator or vTPM daemon.
- `video` - Allows the guest to leverage the host's video capabilities.
- `wayland` - Allows the guest to use the host's Wayland socket.
- `vsock` - Enables use of virtual sockets for the guest.
- `vhost-user` - VirtIO devices which offloads the device implementation to another process through the vhost-user protocol.
  - vmm side: Shares its virtqueues.
  - device side: Consumes virtqueues.

Block

crosvm supports virtio-block device that works as a disk for the guest.

First, create a ext4 (or whatever file system you want) disk file.

```
fallocate -l 1G disk.img
mkfs.ext4 disk.img
```

Then, pass it with `--block` flag so the disk will be exposed as `/dev/vda`, `/dev/vdb`, etc. The device can be mounted with the `mount` command.

```
crosvm run \
  --block disk.img
  ... # usual crosvm args
```

To expose the block device as a read-only disk, you can add the `ro` flag after the disk image path:

```
crosvm run \
  --block disk.img,ro
  ... # usual crosvm args
```

Rootfs

If you use a block device as guest's rootfs, you can add the `root` flag to the `--block` parameter:

```
crosvm run \
  --block disk.img,root
  ... # usual crosvm args
```

This flag automatically adds a `root=/dev/vdX` kernel parameter with the corresponding virtio-block device name and read-only (`ro`) or read-write (`rw`) option depending on whether the `ro` flag has also been specified or not.

Options

The `--block` parameter support additional options to enable features and control disk parameters. These may be specified as extra comma-separated `key=value` options appended to the required filename option. For example:

```
crosvm run
  --block disk.img,ro,sparse=false,o_direct=true,block_size=4096,id=MYSERIALNO
  ... # usual crosvm args
```

The available options are documented in the following sections.

Sparse

- Syntax: `sparse=(true|false)`
- Default: `sparse=true`

The `sparse` option controls whether the disk exposes the thin provisioning `discard` command. If `sparse` is set to `true`, the `VIRTIO_BLK_T_DISCARD` request will be available, and it will be translated to the appropriate system call on the host disk image file (for example, `fallocate(FALLOC_FL_PUNCH_HOLE)` for raw disk images on Linux). If `sparse` is set to `false`, the disk will be fully allocated at startup (using `fallocate()` or equivalent on other platforms), and the `VIRTIO_BLK_T_DISCARD` request will not be supported for this device.

`O_DIRECT`

- Syntax: `o_direct=(true|false)`
- Default: `o_direct=false`

The `o_direct` option enables the Linux `O_DIRECT` flag on the underlying disk image, indicating that I/O should be sent directly to the backing storage device rather than using the host page cache. This should only be used with raw disk images, not qcow2 or other formats. The `block_size` option may need to be adjusted to ensure that I/O is sufficiently aligned for the host block device and filesystem requirements.

Block size

- Syntax: `block_size=BYTES`
- Default: `block_size=512`

The `block_size` option overrides the reported block size (also known as sector size) of the virtio-block device. This should be a power of two larger than or equal to 512.

ID

- Syntax: `id=DISK_ID`
- Default: No identifier

The `id` option provides the virtio-block device with a unique identifier. The `DISK_ID` string must be 20 or fewer ASCII printable characters. The `id` may be used by the guest environment to uniquely

identify a specific block device rather than making assumptions about block device names.

The Linux virtio-block driver exposes the disk identifer in a `sysfs` file named `serial`; an example path looks like `/sys/devices/pci0000:00/0000:00:02.0/virtio1/block/vda/serial` (the PCI address may differ depending on which other devices are enabled).

Resizing

The crosvm block device supports run-time resizing. This can be accomplished by starting crosvm with the `-s` control socket, then using the `crosvm disk` command to send a resize request:

`crosvm disk resize DISK_INDEX NEW_SIZE VM_SOCKET`

- `DISK_INDEX`: 0-based index of the block device (counting all `--block` in order).
- `NEW_SIZE`: desired size of the disk image in bytes.
- `VM_SOCKET`: path to the VM control socket specified when running crosvm (`-s`/`--socket` option).

For example:

```
# Create a 1 GiB disk image
truncate -s 1G disk.img

# Run crosvm with a control socket
crosvm run \
  --block disk.img,sparse=false \
  -s /tmp/crosvm.sock \
  ... # other crosvm args

# In another shell, extend the disk image to 2 GiB.
crosvm disk resize \
  0 \
  $((2 * 1024 * 1024 * 1024)) \
  /tmp/crosvm.sock

# The guest OS should recognize the updated size and log a message:
#   virtio_blk virtio1: [vda] new size: 4194304 512-byte logical blocks (2.15 GB/2.00 GiB)
```

The `crosvm disk resize` command only resizes the block device and its backing disk image. It is the responsibility of the VM socket user to perform any partition table or filesystem resize operations, if required.

Network

The most convenient way to provide a network device to a guest is to setup a persistent TAP interface on the host. This section will explain how to do this for basic IPv4 connectivity.

```
sudo ip tuntap add mode tap user $USER vnet_hdr crosvm_tap
sudo ip addr add 192.168.10.1/24 dev crosvm_tap
sudo ip link set crosvm_tap up
```

These commands create a TAP interface named `crosvm_tap` that is accessible to the current user, configure the host to use the IP address `192.168.10.1`, and bring the interface up.

The next step is to make sure that traffic from/to this interface is properly routed:

```
sudo sysctl net.ipv4.ip_forward=1
# Network interface used to connect to the internet.
HOST_DEV=$(ip route get 8.8.8.8 | awk -- '{printf $5}')
sudo iptables -t nat -A POSTROUTING -o "${HOST_DEV}" -j MASQUERADE
sudo iptables -A FORWARD -i "${HOST_DEV}" -o crosvm_tap -m state --state
RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i crosvm_tap -o "${HOST_DEV}" -j ACCEPT
```

The interface is now configured and can be used by crosvm:

```
crosvm run \
  ...
  --net tap-name=crosvm_tap \
  ...
```

Provided the guest kernel had support for `VIRTIO_NET`, the network device should be visible and configurable from the guest:

```
# Replace with the actual network interface name of the guest
# (use "ip addr" to list the interfaces)
GUEST_DEV=enp0s5
sudo ip addr add 192.168.10.2/24 dev "${GUEST_DEV}"
sudo ip link set "${GUEST_DEV}" up
sudo ip route add default via 192.168.10.1
# "8.8.8.8" is chosen arbitrarily as a default, please replace with your local (or
preferred global)
# DNS provider, which should be visible in `/etc/resolv.conf` on the host.
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
```

These commands assign IP address `192.168.10.2` to the guest, activate the interface, and route all network traffic to the host. The last line also ensures DNS will work.

Please refer to your distribution's documentation for instructions on how to make these settings persistent for the host and guest if desired.

Balloon

crosvm supports virtio-balloon for managing guest memory.

How to control the balloon size

When running a VM, specify `VM_SOCKET` with `-s` option. (example: `/run/crosvm.sock`)

```
crosvm run \
    -s ${CROSVM_SOCKET} \
    # usual crosvm args
    /path/to/bzImage
```

Then, open another terminal and specify the balloon size in bytes with `crosvm balloon` command.

```
crosvm balloon 4096 ${CROSVM_SOCKET}
```

Note: The size of balloon is managed in 4096 bytes units. The specified value will be rounded down to a multiple of 4096 bytes.

You can confirm the balloon size with `crosvm balloon_stats` command.

```
crosvm balloon_stats ${CROSVM_SOCKET}
```

Vsock device

crosvm supports virtio-vsock device for communication between the host and a guest VM.

Assign a context id to a guest VM by passing it with `--cid` flag.

```
GUEST_CID=3

crosvm run \
  --cid "${GUEST_CID}" \
  <usual crosvm arguments>
  /path/to/bzImage
```

Then, the guest and the host can communicate with each other via vsock. Host always has 2 as its context id.

crosvm assumes that the host has a vsock device at `/dev/vhost-vsock`. If you want to use a device at a different path or one given as an fd, you can use `--vhost-vsock-device` flag or `--vhost-vsock-fd` flag respectively.

Example usage

At host shell:

```
PORT=11111

# Listen at host
ncat -l --vsock ${PORT}
```

At guest shell:

```
HOST_CID=2
PORT=11111

# Make a connection to the host
ncat --vsock ${HOST_CID} ${PORT}
```

If a vsock device is configured properly in the guest VM, a connection between the host and the guest can be established and packets can be sent from both side. In the above example, your inputs to a shell on one's side should be shown at the shell on the other side if a connection is successfully established.

Pmem

crosvm supports `virtio-pmem` to provide a virtual device emulating a byte-addressable persistent memory device. The disk image is provided to the guest using a memory-mapped view of the image file, and this mapping can be directly mapped into the guest's address space if the guest operating system and filesystem support DAX.

Pmem devices may be added to crosvm using the `--pmem-device` (read only) or `--rw-pmem-device` (read-write) flag, specifying the filename of the backing image as the parameter.

```
crosvm run \
  --pmem-device disk.img \
  ... # usual crosvm args
```

The Linux virtio-pmem driver can be enabled with the `CONFIG_VIRTIO_PMEM` option. It will expose pmem devices as `/dev/pmem0`, `/dev/pmem1`, etc., which may be mounted like any other block device. A pmem device may also be used as a root filesystem by adding a `root=` kernel command line parameters:

```
crosvm run \
  --pmem-device rootfs.img \
  -p "root=/dev/pmem0 ro" \
  ... # usual crosvm args
```

The advantage of pmem over a regular block device is the potential for less cache duplication; since the guest can directly map pages of the pmem device, it does not need to perform an extra copy into the guest page cache. This can result in lower memory overhead versus `virtio-block` (when not using `O_DIRECT`).

The file backing a persistent memory device is mapped directly into the guest's address space, which means that only the raw disk image format is supported; disk images in qcow2 or other formats may not be used as a pmem device. See the `block` device for an alternative that supports more file formats.

Wayland

If you have a Wayland compositor running on your host, it is possible to display and control guest applications from it. This requires:

- A guest kernel version 5.16 or above with `CONFIG_DRM_VIRTIO_GPU` enabled,
- The `sommelier` Wayland proxy in your guest image.

This section will walk you through the steps needed to get this to work.

Guest kernel requirements

Wayland support on crosvm relies on virtio-gpu contexts, which have been introduced in Linux 5.16. Make sure your guest kernel is either this version or a more recent one, and that `CONFIG_DRM_VIRTIO_GPU` is enabled in your kernel configuration.

Crosvm requirements

Wayland forwarding requires the GPU feature and the virtio-gpu cross domain mode to be enabled.

```
cargo build --features "gpu"
```

Building sommelier

Sommelier is a proxy Wayland compositor that forwards the Wayland protocol from a guest to a compositor running on the host through the guest GPU device. As it is not a standard tool, we will have to build it by ourselves. It is recommended to do this from the guest with networking enabled.

Clone ChromeOS' `platform2` repository, which contains the source for sommelier:

```
git clone https://chromium.googlesource.com/chromiumos/platform2
```

Go into the sommelier directory and prepare for building:

```
cd platform2/vm_tools/sommelier/
meson setup build -Dwith_tests=false
```

This setup step will check for all libraries required to build sommelier. If some are missing, install them using your guest's distro package manager and re-run `meson setup` until it passes.

Finally, build sommelier and install it:

```
meson compile -C build
sudo meson install -C build
```

This last step will put the `sommelier` binary into `/usr/local/bin`.

Running guest Wayland apps

Crosvm can connect to a running Wayland server (e.g. weston) on the host and forward the protocol from all Wayland guest applications to it. To enable this you need to know the socket of the Wayland server running on your host - typically it would be `$XDG_RUNTIME_DIR/wayland-0`.

Once you have confirmed the socket, create a GPU device and enable forwarding by adding the `--gpu=context-types=cross-domain --wayland-sock $XDG_RUNTIME_DIR/wayland-0` arguments to your crosvm command-line. Other context types may be also enabled for those interested in 3D acceleration.

You can now run Wayland clients through sommelier, e.g:

```
sommelier --virtgpu-channel weston-terminal
```

Or

```
sommelier --virtgpu-channel gedit
```

Applications started that way should appear on and be controllable from the Wayland server running on your host.

The `--virtgpu-channel` option is currently necessary for sommelier to work with the setup of this document, but will likely not be required in the future.

If you have `Xwayland` installed in the guest you can also run X applications:

```
sommelier -X --xwayland-path=/usr/bin/Xwayland xeyes
```

Video (experimental)

The virtio video decoder and encoder devices allow a guest to leverage the host's hardware-accelerated video decoding and encoding capabilities. The specification (v3, v5) for these devices is still a work-in-progress, so testing them requires an out-of-tree kernel driver on the guest.

The virtio-video host device uses backends to perform the actual decoding. The currently supported backends are:

- `libvda`, a hardware-accelerated backend that supports both decoding and encoding by delegating the work to a running instance of Chrome. It can only be built and used in a ChromeOS environment.
- `ffmpeg`, a software-based backend that supports encoding and decoding. It exists to make testing and development of virtio-video easier, as it does not require any particular hardware and is based on a reliable codec library.

The rest of this document will solely focus on the `ffmpeg` backend. More accelerated backends will be added in the future.

Guest kernel requirements

The `virtio_video` branch of this kernel git repository contains a work-in-progress version of the `virtio-video` guest kernel driver, based on a (hopefully) recent version of mainline Linux. If you use this as your guest kernel, the `virtio_video_defconfig` configuration should allow you to easily boot from crosvm, with the video (and a few other) virtio devices support built-in.

Quick building guide after checking out this branch:

```
mkdir build_crosvm_x86
make O=build_crosvm_x86 virtio_video_defconfig
make O=build_crosvm_x86 -j16
```

The resulting kernel image that can be passed to `crosvm` will be in `build_crosvm_x86/arch/x86/boot/bzImage`.

Crosvm requirements

The virtio-video support is experimental and needs to be opted-in through the `"video-decoder"` or `"video-encoder"` Cargo feature. In the instruction below we'll be using the FFmpeg backend which requires the `"ffmpeg"` feature to be enabled as well.

The following example builds crosvm with FFmpeg encoder and decoder backend support:

```
cargo build --features "video-encoder,video-decoder,ffmpeg"
```

To enable the **decoder** device, start crosvm with the `--video-decoder=ffmpeg` command-line argument:

```
crosvm run --disable-sandbox --video-decoder=ffmpeg -c 4 -m 2048 --block
/path/to/disk.img,root --serial type=stdout,hardware=virtio-
console,console=true,stdin=true /path/to/bzImage
```

Alternatively, to enable the **encoder** device, start crosvm with the `--video-encoder=ffmpeg` command-line argument:

```
crosvm run --disable-sandbox --video-encoder=ffmpeg -c 4 -m 2048 --block
/path/to/disk.img,root --serial type=stdout,hardware=virtio-
console,console=true,stdin=true /path/to/bzImage
```

If the guest kernel includes the virtio-video driver, then the device should be probed and show up.

Testing the device from the guest

Video capabilities are exposed to the guest using V4L2. The encoder or decoder device should appear as `/dev/videoX`, probably `/dev/video0` if there are no additional V4L2 devices.

Checking capabilities and formats

`v4l2-ctl`, part of the `v4l-utils` package, can be used to test the device's existence.

Example output for the decoder is shown below.

```
v4l2-ctl -d/dev/video0 --info
Driver Info:
        Driver name      : virtio-video
        Card type        : ffmpeg
        Bus info         : virtio:stateful-decoder
        Driver version   : 5.17.0
        Capabilities     : 0x84204000
                Video Memory-to-Memory Multiplanar
                Streaming
                Extended Pix Format
                Device Capabilities
        Device Caps      : 0x04204000
                Video Memory-to-Memory Multiplanar
                Streaming
                Extended Pix Format
```

Note that the `Card type` is `ffmpeg`, indicating that decoding will be performed in software on the host. We can then query the support input (`OUTPUT` in V4L2-speak) formats, i.e. the encoded formats we can send to the decoder:

```
v4l2-ctl -d/dev/video0 --list-formats-out
ioctl: VIDIOC_ENUM_FMT
        Type: Video Output Multiplanar

        [0]: 'VP90' (VP9, compressed)
        [1]: 'VP80' (VP8, compressed)
        [2]: 'HEVC' (HEVC, compressed)
        [3]: 'H264' (H.264, compressed)
```

Similarly, you can check the supported output (or CAPTURE) pixel formats for decoded frames:

```
v4l2-ctl -d/dev/video0 --list-formats
ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture Multiplanar

        [0]: 'NV12' (Y/CbCr 4:2:0)
```

Test decoding with ffmpeg

FFmpeg can be used to decode video streams with the virtio-video device.

Simple VP8 stream:

```
wget https://github.com/chromium/chromium/raw/main/media/test/data/test-25fps.vp8
ffmpeg -codec:v vp8_v4l2m2m -i test-25fps.vp8 test-25fps-%d.png
```

This should create 250 PNG files each containing a decoded frame from the stream.

WEBM VP9 stream:

```
wget https://test-
videos.co.uk/vids/bigbuckbunny/webm/vp9/720/Big_Buck_Bunny_720_10s_1MB.webm
ffmpeg -codec:v vp9_v4l2m2m -i Big_Buck_Bunny_720_10s_1MB.webm Big_Buck_Bunny-%d.png
```

Should create 300 PNG files at 720p resolution.

Test decoding with v4l2r

The v4l2r Rust crate also features an example program that can use this driver to decode simple H.264 streams:

```
git clone https://github.com/Gnurou/v4l2r
cd v4l2r
wget https://github.com/chromium/chromium/raw/main/media/test/data/test-25fps.h264
cargo run --example simple_decoder test-25fps.h264 /dev/video0 --input_format h264 --
save test-25fps.nv12
```

This will decode `test-25fps.h264` and write the raw decoded frames in `NV12` format into `test-25fps.nv12`. You can check the result with e.g. YUView.

Test encoding with ffmpeg

FFmpeg can be used to encode video streams with the virtio-video device.

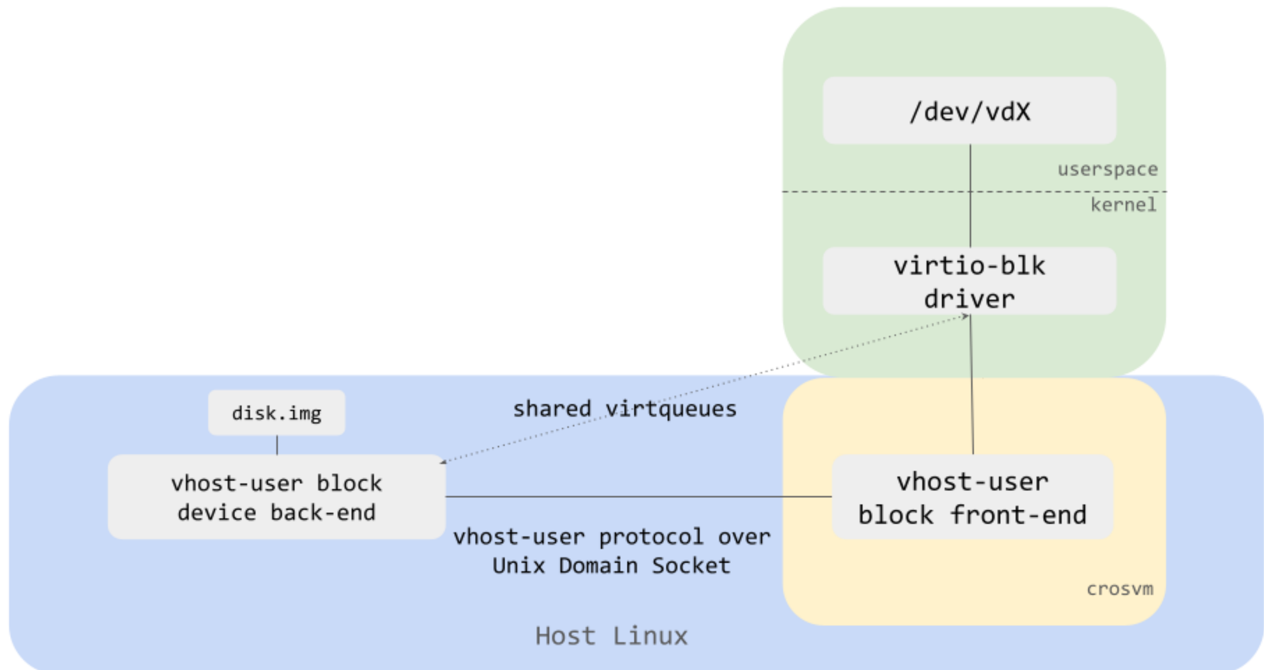
The following examples generates a test clip through libavfilter and encode it using the virtual H.264, H.265 and VP8 encoder, respectively. (VP9 v4l2m2m support is missing in FFmpeg for some reason.)

```
# H264
ffmpeg -f lavfi -i smptebars=duration=10:size=640x480:rate=30 \
  -pix_fmt nv12 -c:v h264_v4l2m2m smptebars.h264.mp4
# H265
ffmpeg -f lavfi -i smptebars=duration=10:size=640x480:rate=30 \
  -pix_fmt yuv420p -c:v hevc_v4l2m2m smptebars.h265.mp4
# VP8
ffmpeg -f lavfi -i smptebars=duration=10:size=640x480:rate=30 \
  -pix_fmt yuv420p -c:v vp8_v4l2m2m smptebars.vp8.webm
```

Vhost-user devices

Crosvm supports vhost-user devices for most virtio devices (block, net, etc) so that device emulation can be done outside of the main vmm process.

Here is a diagram showing how vhost-user block device back-end and a vhost-user block front-end in crosvm VMM work together.



How to run

Let's take a block device as an example and see how to start vhost-user devices.

First, start vhost-user block backend with `crosvm devices` command, which waits for a vmm process connecting to the socket.

```
# One-time commands to create a disk image.
fallocate -l 1G disk.img
mkfs.ext4 disk.img

VHOST_USER_SOCK=/tmp/vhost-user.socket

# Start vhost-user block backend listening on $VHOST_USER_SOCK
crosvm devices --block vhost=${VHOST_USER_SOCK},path=disk.img
```

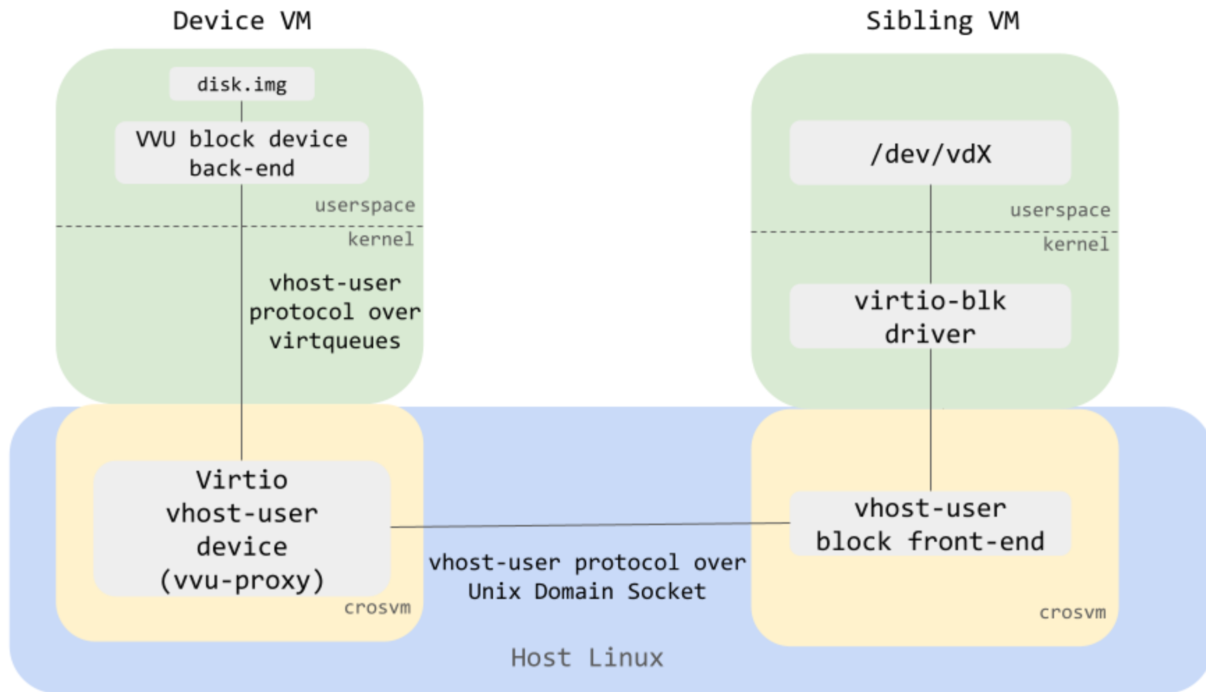
Then, open another terminal and start a vmm process with `--vhost-user-blk` flag.

```
crosvm run \
  --vhost-user-blk "${VHOST_USER_SOCK}" \
  <usual crosvm arguments>
  /path/to/bzImage
```

As a result, `disk.img` should be exposed as `/dev/vda` just like with `--block disk.img`.

Virtio Vhost-User device (VVU)

Crosvm also supports the virtio vhost-user (VVU) device to run a vhost-user device back-end inside of another VM's guest. The following diagram shows how VVU works for virtio-block.



The "virtio vhost-user device", which is also called "vvu-proxy", is a virtio PCI device that works as a proxy of vhost-user messages between the vhost-user device back-end in the guest of a VM (device VM) and the vhost-user front-end in another VM (sibling VM).

How to run

Let's take a block device as an example and see how to start VVU devices.

First, start a device VM with a usual `crosvm run` command. At this time, put a crosvm binary in the guest in some way. (e.g. putting it in a disk, sharing the host's crosvm with virtiofs, building crosvm in the guest, etc). Also, make sure that the guest kernel is configured properly with virtio and vfio features enabled (see caveat).

```
# On the host.

VHOST_USER_SOCK=/tmp/vhost-user.socket

# Specify the PCI address that the VVU proxy device will be allocated.
# If you don't pass `addr=` as an argument of `--vvu-proxy` below, crosvm will
# allocate it to the first available address.
VVU_PCI_ADDR="0000:00:10.0"

# Start the device VM with '-p "vfio_iommu_type1.allow_unsafe_interrupts=1"'.
crosvm run \
  --vvu-proxy "${VHOST_USER_SOCK},addr=${VVU_PCI_ADDR}" \
  -p "vfio_iommu_type1.allow_unsafe_interrupts=1" \
  -m 4096 \ # Make sure that the device kernel has enough memory to be used
  ... # usual crosvm args
  /path/to/bzImage
```

Then you can check that the VVU proxy device is allocated at the specified address by running `lspci` in the guest.

```
# Inside of the device VM guest.

lspci -s $VVU_PCI_ADDR
# Expected output:
# > 00:10.0 Unclassified device [00ff]: Red Hat, Inc. Device 107d (rev 01)
# '107d' is the device ID for the VVU proxy device.
```

After that you need to make sure that the VVU device is bound to vfio_pci driver by manipulating sysfs.

```
# Inside of the device VM guest.
basename `readlink /sys/bus/pci/devices/$VVU_PCI_ADDR/driver`
# If that shows vfio-pci you are done, otherwise you need to rebind
# the device to the right driver.
echo "vfio-pci" > /sys/bus/pci/devices/$VVU_PCI_ADDR/driver_override
echo "$VVU_PCI_ADDR" > /sys/bus/pci/devices/$VVU_PCI_ADDR/driver/unbind
echo "$VVU_PCI_ADDR" > /sys/bus/pci/drivers/vfio-pci/bind
basename `readlink /sys/bus/pci/devices/$VVU_PCI_ADDR/driver`
# This should show "vfio-pci" now.
```

Then, start a VVU block device backend in the guest that you just started. Although the command `crosvm device` is the same as vhost-user's example, you need to use the `--vfio` flag instead of the `--socket` flag.

```
# Inside of the device VM guest.

crosvm device block \
  --vfio ${VVU_PCI_ADDR} \
  --file disk.img
```

Finally, open another terminal and start a vmm process with `--vhost-user-blk` flag on the host. The current implementation of crosvm only allows a sibling VM to have a smaller memory size than the device VM, so make sure to specify the memory size correctly.

```
# On the host, start a sibling VM. This can be done in the same way as the vhost-user
block front-end.

crosvm run \
  --vhost-user-blk ${VHOST_USER_SOCK} \
  -m 512 \ # Make sure that the sibling VM does not have same or more memory than the
device VM
  ... # usual crosvm args
  /path/to/bzImage
```

As a result, `disk.img` in the device VM should be exposed as `/dev/vda` in the guest of the sibling VM.

Caveats

- In order to use the VVU feature, the Device VM kernel is required to be configured with couple of vfio features. Note that the name of the config may vary depending on the version of the kernel. We expect that the readers follow the instructions in this page to create a custom kernel. In addition to the instruction, the required configurations in the Linux Kernel version 5.10 are:
  - CONFIG_ACPI
  - CONFIG_VFIO
  - CONFIG_VFIO_PCI
  - CONFIG_VIRTIO_IOMMU
- Currently, the sibling VM is required to have less memory than the device VM. Make sure that the memory size is explicitly defined when starting the VM for both device and sibling VMs.

Crosvm System Integration

The following sections describe how crosvm is integrated into other projects.

Crosvm on ChromeOS

A copy of crosvm is included in the ChromeOS source tree at chromiumos/platform/crosvm, which is referred to as **downstream** crosvm.

All crosvm development is happening **upstream** at crosvm/crosvm. Changes from upstream crosvm are regularly merged with ChromeOS's downstream crosvm.

The merge process.

A crosvm bot will regularly generate automated commits that merge upstream crosvm into downstream. These commits can be found in gerrit.

The crosvm team is submitting these merges through the ChromeOS CQ regularly, which happens **roughly once per week**, but time can vary depending on CQ health.

Googlers can find more information on the merge process at go/crosvm/playbook

Building crosvm for ChromeOS

crosvm on ChromeOS is usually built with Portage, so it follows the same general workflow as any `cros_workon` package. The full package name is `chromeos-base/crosvm`.

The developer guide section on Make your Changes applies to crosvm as well. You can build crosvm with `cros_workon_make`:

```
cros_workon --board=${BOARD} start crosvm
cros_workon_make --board=${BOARD} crosvm
```

Deploy it via `cros deploy`:

```
cros_workon_make --board=${BOARD} --install crosvm
cros deploy ${IP} crosvm
```

Iterative test runs can be done as well:

```
cros_workon_make --board=${BOARD} --test crosvm
```

Warning: `cros_workon_make` patches the local Cargo.toml file. Please do not submit these changes.

Rebuilding all crosvm dependencies

Crosvm has a lot of rust dependencies that are installed into a registry inside cros_sdk. After a `repo sync` these can be out of date, causing compilation issues. To make sure all dependencies are up to date, run:

```
emerge-${BOARD} --update --deep -j$(nproc) crosvm
```

Building crosvm for Linux

`emerge` and `cros_workon_make` workflows can be quite slow to work with, hence a lot of developers prefer to use standard cargo workflows used upstream.

Just make sure to initialize git submodules (`git submodules update --init`), which is not done by repo. After that, you can use the workflows as outlined in Building Crosvm **outside** of cros_sdk.

Note: You can **not** build or test ChromeOS specific features this way.

Submitting Changes

All changes to crosvm are made upstream, using the same process outlined in Contributing. It is recommended to use the Building crosvm for Linux setup above to run upstream presubmit checks / formatting tools / etc when submitting changes.

Code submitted upstream is tested on linux, but not on ChromeOS devices. Changes will only be tested on the ChromeOS CQ when they go through the merge process.

Has my change landed in ChromeOS (Googlers only)?

You can use the crosland tool to check in which ChromeOS version your changes have been merged into the chromiumos/platform/crosvm repository.

The merge will also contain all `BUG=` references that will notify your bugs about when the change is submitted.

For more details on the process, please see go/crosvm-playbook (Google only).

Cq-Depend

We cannot support Cq-Depend to sychronize changes with other ChromeOS repositories. Please try to make changes in a backwards compatible way to allow them to be submitted independently.

If it cannot be avoided at all, please follow this process:

1. Upload your change to upstream crosvm and get it reviewed. Do not submit it yet.
2. Upload the change to chromiumos/platform/crosvm as well.
3. Use Cq-Depend on the ChromeOS changes and submit it via the CQ.
4. After the changes landed in ChromeOS, land them upstream as well.

Cherry-picking

If you need your changes faster than the usual merge frequency, please follow this process:

1. Upload and submit your change to upstream crosvm.
2. Upload the change to chromiumos/platform/crosvm as well.
3. Submit as usual through the CQ.

Never submit code just to ChromeOS, as it will cause upstream to diverge and result in merge conflicts down the road.

Architecture

The principle characteristics of crosvm are:

- A process per virtual device, made using fork
- Each process is sandboxed using minijail
- Takes full advantage of KVM and low-level Linux syscalls, and so only runs on Linux
- Written in Rust for security and safety

A typical session of crosvm starts in `main.rs` where command line parsing is done to build up a `Config` structure. The `Config` is used by `run_config` in `linux/mod.rs` to setup and execute a VM. Broken down into rough steps:

1. Load the linux kernel from an ELF file.
2. Create a handful of control sockets used by the virtual devices.
3. Invoke the architecture specific VM builder `Arch::build_vm` (located in `x86_64/src/lib.rs` or `aarch64/src/lib.rs`).
4. `Arch::build_vm` will itself invoke the provided `create_devices` function from `linux/mod.rs`
5. `create_devices` creates every PCI device, including the virtio devices, that were configured in `Config`, along with matching minijail configs for each.
6. `Arch::assign_pci_addresses` assigns an address to each PCI device, prioritizing devices that report a preferred slot by implementing the `PciDevice` trait's `preferred_address` function.
7. `Arch::generate_pci_root`, using a list of every PCI device with optional `Minijail`, will finally jail the PCI devices and construct a `PciRoot` that communicates with them.
8. Once the VM has been built, it's contained within a `RunnableLinuxVm` object that is used by the VCPUs and control loop to service requests until shutdown.

Forking

During the device creation routine, each device will be created and then wrapped in a `ProxyDevice` which will internally `fork` (but not `exec`) and minijail the device, while dropping it for the main process. The only interaction that the device is capable of having with the main process is via the proxied trait methods of `BusDevice`, shared memory mappings such as the guest memory, and file descriptors that were specifically allowed by that device's security policy. This can lead to some surprising behavior to be aware of such as why some file descriptors which were once valid are now invalid.

Sandboxing Policy

Every sandbox is made with minijail and starts with `create_base_minijail` in `linux/jail_helpers.rs` which set some very restrictive settings. Linux namespaces and seccomp filters are used extensively. Each seccomp policy can be found under

`seccomp/{arch}/{device}.policy` and should start by `@include`-ing the `common_device.policy`. With the exception of architecture specific devices (such as `Pl030` on ARM or `I8042` on x86_64), every device will need a different policy for each supported architecture.

The VM Control Sockets

For the operations that devices need to perform on the global VM state, such as mapping into guest memory address space, there are the vm control sockets. There are a few kinds, split by the type of request and response that the socket will process. This also proves basic security privilege separation in case a device becomes compromised by a malicious guest. For example, a rogue device that is able to allocate MSI routes would not be able to use the same socket to (de)register guest memory. During the device initialization stage, each device that requires some aspect of VM control will have a constructor that requires the corresponding control socket. The control socket will get preserved when the device is sandboxed and the other side of the socket will be waited on in the main process's control loop.

The socket exposed by crosvm with the `--socket` command line argument is another form of the VM control socket. Because the protocol of the control socket is internal and unstable, the only supported way of using that resulting named unix domain socket is via crosvm command line subcommands such as `crosvm stop`.

GuestMemory

`GuestMemory` and its friends `VolatileMemory`, `VolatileSlice`, `MemoryMapping`, and `SharedMemory`, are common types used throughout crosvm to interact with guest memory. Know which one to use in what place using some guidelines

- `GuestMemory` is for sending around references to all of the guest memory. It can be cloned freely, but the underlying guest memory is always the same. Internally, it's implemented using `MemoryMapping` and `SharedMemory`. Note that `GuestMemory` is mapped into the host address space (for non-protected VMs), but it is non-contiguous. Device memory, such as mapped DMA-Bufs, are not present in `GuestMemory`.
- `SharedMemory` wraps a `memfd` and can be mapped using `MemoryMapping` to access its data. `SharedMemory` can't be cloned.
- `VolatileMemory` is a trait that exposes generic access to non-contiguous memory. `GuestMemory` implements this trait. Use this trait for functions that operate on a memory space but don't necessarily need it to be guest memory.
- `VolatileSlice` is analogous to a Rust slice, but unlike those, a `VolatileSlice` has data that changes asynchronously by all those that reference it. Exclusive mutability and data synchronization are not available when it comes to a `VolatileSlice`. This type is useful for functions that operate on contiguous shared memory, such as a single entry from a scatter gather table, or for safe wrappers around functions which operate on pointers, such as a `read` or `write` syscall.

- `MemoryMapping` is a safe wrapper around anonymous and file mappings. Provides RAII and does munmap after use. Access via Rust references is forbidden, but indirect reading and writing is available via `VolatileSlice` and several convenience functions. This type is most useful for mapping memory unrelated to `GuestMemory`.

See memory layout for details how crosvm arranges the guest address space.

Device Model

`Bus`/`BusDevice`

The root of the crosvm device model is the `Bus` structure and its friend the `BusDevice` trait. The `Bus` structure is a virtual computer bus used to emulate the memory-mapped I/O bus and also I/O ports for x86 VMs. On a read or write to an address on a VM's bus, the corresponding `Bus` object is queried for a `BusDevice` that occupies that address. `Bus` will then forward the read/write to the `BusDevice`. Because of this behavior, only one `BusDevice` may exist at any given address. However, a `BusDevice` may be placed at more than one address range. Depending on how a `BusDevice` was inserted into the `Bus`, the forwarded read/write will be relative to 0 or to the start of the address range that the `BusDevice` occupies (which would be ambiguous if the `BusDevice` occupied more than one range).

Only the base address of a multi-byte read/write is used to search for a device, so a device implementation should be aware that the last address of a single read/write may be outside its address range. For example, if a `BusDevice` was inserted at base address 0x1000 with a length of 0x40, a 4-byte read by a VCPU at 0x39 would be forwarded to that `BusDevice`.

Each `BusDevice` is reference counted and wrapped in a mutex, so implementations of `BusDevice` need not worry about synchronizing their access across multiple VCPUs and threads. Each VCPU will get a complete copy of the `Bus`, so there is no contention for querying the `Bus` about an address. Once the `BusDevice` is found, the `Bus` will acquire an exclusive lock to the device and forward the VCPU's read/write. The implementation of the `BusDevice` will block execution of the VCPU that invoked it, as well as any other VCPU attempting access, until it returns from its method.

Most devices in crosvm do not implement `BusDevice` directly, but some are examples are `i8042` and `Serial`. With the exception of PCI devices, all devices are inserted by architecture specific code (which may call into the architecture-neutral `arch` crate). A `BusDevice` can be proxied to a sandboxed process using `ProxyDevice`, which will create the second process using a fork, with no exec.

`PciConfigIo`/`PciConfigMmio`

In order to use the more complex PCI bus, there are a couple adapters that implement `BusDevice` and call into a `PciRoot` with higher level calls to `config_space_read`/`config_space_write`. The `PciConfigMmio` is a `BusDevice` for insertion into the MMIO `Bus` for ARM devices. For x86_64,

`PciConfigIo` is inserted into the I/O port `Bus`. There is only one implementation of `PciRoot` that is used by either of the `PciConfig*` structures. Because these devices are very simple, they have very little code or state. They aren't sandboxed and are run as part of the main process.

`PciRoot`/`PciDevice`/`VirtioPciDevice`

The `PciRoot`, analogous to `BusDevice` for `Bus`s, contains all the `PciDevice` trait objects. Because of a shortcut (or hack), the `ProxyDevice` only supports jailing `BusDevice` traits. Therefore, `PciRoot` only contains `BusDevice`s, even though they also implement `PciDevice`. In fact, every `PciDevice` also implements `BusDevice` because of a blanket implementation (`impl<T: PciDevice> BusDevice for T { … }`). There are a few PCI related methods in `BusDevice` to allow the `PciRoot` to still communicate with the underlying `PciDevice` (yes, this abstraction is very leaky). Most devices will not implement `PciDevice` directly, instead using the `VirtioPciDevice` implementation for virtio devices, but the xHCI (USB) controller is an example that implements `PciDevice` directly. The `VirtioPciDevice` is an implementation of `PciDevice` that wraps a `VirtioDevice`, which is how the virtio specified PCI transport is adapted to a transport agnostic `VirtioDevice` implementation.

`VirtioDevice`

The `VirtioDevice` is the most widely implemented trait among the device traits. Each of the different virtio devices (block, rng, net, etc.) implement this trait directly and they follow a similar pattern. Most of the trait methods are easily filled in with basic information about the specific device, but `activate` will be the heart of the implementation. It's called by the virtio transport after the guest's driver has indicated the device has been configured and is ready to run. The virtio device implementation will receive the run time related resources (`GuestMemory`, `Interrupt`, etc.) for processing virtio queues and associated interrupts via the arguments to `activate`, but `activate` can't spend its time actually processing the queues. A VCPU will be blocked as long as `activate` is running. Every device uses `activate` to launch a worker thread that takes ownership of run time resources to do the actual processing. There is some subtlety in dealing with virtio queues, so the smart thing to do is copy a simpler device and adapt it, such as the rng device (`rng.rs`).

Communication Framework

Because of the multi-process nature of crosvm, communication is done over several IPC primitives. The common ones are shared memory pages, unix sockets, anonymous pipes, and various other file descriptor variants (DMA-buf, eventfd, etc.). Standard methods (`read`/`write`) of using these primitives may be used, but crosvm has developed some helpers which should be used where applicable.

`WaitContext`

Most threads in crosvm will have a wait loop using a `WaitContext`, which is a wrapper around a `epoll` on Linux and `WaitForMultipleObjects` on Windows. In either case, waitable objects can be added to the context along with an associated token, whose type is the type parameter of `WaitContext`. A call to the `wait` function will block until at least one of the waitable objects has become signaled and will return a collection of the tokens associated with those objects. The tokens used with `WaitContext` must be convertible to and from a `u64`. There is a custom derive `#[derive(EventToken)]` which can be applied to an `enum` declaration that makes it easy to use your own enum in a `WaitContext`.

Linux Platform Limitations

The limitations of `WaitContext` on Linux are the same as the limitations of `epoll`. The same FD can not be inserted more than once, and the FD will be automatically removed if the process runs out of references to that FD. A `dup` / `fork` call will increment that reference count, so closing the original FD will not actually remove it from the `WaitContext`. It is possible to receive tokens from `WaitContext` for an FD that was closed because of a race condition in which an event was registered in the background before the `close` happened. Best practice is to keep an FD open and remove it from the `WaitContext` before closing it so that events associated with it can be reliably eliminated.

`serde` with Descriptors

Using raw sockets and pipes to communicate is very inconvenient for rich data types. To help make this easier and less error prone, crosvm uses the `serde` crate. To allow transmitting types with embedded descriptors (FDs on Linux or HANDLEs on Windows), a module is provided for sending and receiving descriptors alongside the plain old bytes that serde consumes.

Code Map

Source code is organized into crates, each with their own unit tests.

- `./src/` - The top-level binary front-end for using crosvm.
- `aarch64` - Support code specific to 64 bit ARM architectures.
- `base` - Safe wrappers for system facilities which provides cross-platform-compatible interfaces.
- `bin` - Scripts for code health such as wrappers of `rustfmt` and `clippy`.
- `ci` - Scripts for continuous integration.
- `cros_async` - Runtime for async/await programming. This crate provides a `Future` executor based on `io_uring` and one based on `epoll`.
- `devices` - Virtual devices exposed to the guest OS.
- `disk` - Library to create and manipulate several types of disks such as raw disk, qcow, etc.
- `hypervisor` - Abstract layer to interact with hypervisors. For Linux, this crate is a wrapper of `kvm`.

- `e2e_tests` - End-to-end tests that run a crosvm VM.
- `kernel_loader` - Loads elf64 kernel files to a slice of memory.
- `kvm_sys` - Low-level (mostly) auto-generated structures and constants for using KVM.
- `kvm` - Unsafe, low-level wrapper code for using `kvm_sys`.
- `media/libvda` - Safe wrapper of libvda, a ChromeOS HW-accelerated video decoding/encoding library.
- `net_sys` - Low-level (mostly) auto-generated structures and constants for creating TUN/TAP devices.
- `net_util` - Wrapper for creating TUN/TAP devices.
- `qcow_util` - A library and a binary to manipulate qcow disks.
- `seccomp` - Contains minijail seccomp policy files for each sandboxed device. Because some syscalls vary by architecture, the seccomp policies are split by architecture.
- `sync` - Our version of `std::sync::Mutex` and `std::sync::Condvar`.
- `third_party` - Third-party libraries which we are maintaining on the ChromeOS tree or the AOSP tree.
- `vfio_sys` - Low-level (mostly) auto-generated structures, constants and ioctls for VFIO.
- `vhost` - Wrappers for creating vhost based devices.
- `virtio_sys` - Low-level (mostly) auto-generated structures and constants for interfacing with kernel vhost support.
- `vm_control` - IPC for the VM.
- `vm_memory` - Vm-specific memory objects.
- `x86_64` - Support code specific to 64 bit intel machines.

Contributing

Intro

This article goes into detail about multiple areas of interest to contributors, which includes reviewers, developers, and integrators who each share an interest in guiding crosvm's direction.

Bug Reports

We use Google issue tracker. Please use the public crosvm component.

For Googlers: See go/crosvm#filing-bugs.

Philosophy

The following is high level guidance for producing contributions to crosvm.

- Prefer mechanism to policy.
- Use existing protocols when they are adequate, such as virtio.
- Prefer security over code re-use and speed of development.
- Only the version of Rust in use by the ChromeOS toolchain is supported. This is ordinarily the stable version of Rust, but can be behind a version for a few weeks.
- Avoid distribution specific code.

Style guidelines

Formatting

To format all code, crosvm defers to `rustfmt`. In addition, the code adheres to the following rules:

Each `use` statement should import a single item, as produced by `rustfmt` with `imports_granularity=item`. Do not use braces to import multiple items.

The `use` statements for each module should be grouped into blocks separated by whitespace in the order produced by `rustfmt` with `group_imports=StdExternalCrate` and sorted alphabetically:

1. `std`
2. third-party + crosvm crates

3. `crate` + `super`

The import formatting options of `rustfmt` are currently unstable, so these are not enforced automatically. If a nightly Rust toolchain is present, it is possible to automatically reformat the code to match these guidelines by running `tools/fmt --nightly`.

crosvm uses the remain crate to keep error enums sorted, along with the `#[sorted]` attribute to keep their corresponding match statements in the same order.

Unit test code

Unit tests and other highly-specific tests (which may include some small, but not all, integration tests) should be written differently than how non-test code is written. Tests prevent regressions from being committed, show how APIs can be used, and help with understanding bugs in code. That means tests must be clear both now and in the future to a developer with low familiarity of the code under test. They should be understandable by reading from top to bottom without referencing any other code. Towards these goals, tests should:

- To a reasonable extent, be structured as Arrange-Act-Assert.
- Test the minimum number of behaviors in a single test. Make separate tests for separate behavior.
- Avoid helper methods that send critical inputs or assert outputs within the helper itself. It should be easy to read a test and determine the critical inputs/outputs without digging through helper methods. Setup common to many tests is fine to factor out, but lean toward duplicating code if it aids readability.
- Avoid branching statements like conditionals and loops (which can make debugging more difficult).
- Document the reason constants were chosen in the test, including if they were picked arbitrarily such that in the future, changing the value is okay. (This can be done with constant variable names, which is ideal if the value is used more than once, or in a comment.)
- Name tests to describe what is being tested and the expected outcome, for example `test_foo_invalid_bar_returns_baz`.

Less-specific tests, such as most integration tests and system tests, are more likely to require obfuscating work behind helper methods. It is still good to strive for clarity and ease of debugging in those tests, but they do not need to follow these guidelines.

Contributing Code

Prerequisites

You need to set up a user account with gerrit. Once logged in, you can obtain HTTP Credentials to set up git to upload changes.

Once set up, run `./tools/cl` to install the gerrit commit message hook. This will insert a unique "Change-Id" into all commit messages so gerrit can identify changes.

Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License Agreement (CLA). You (or your employer) retain the copyright to your contribution; this simply gives us permission to use and redistribute your contributions as part of the project. Head over to https://cla.developers.google.com/ to see your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Commit Messages

As for commit messages, we follow ChromeOS's guideline in general.

Here is an example of a good commit message:

```
devices: vhost: user: vmm: Add Connection type

This abstracts away the cross-platform differences: cfg(unix) uses a
Unix domain stream socket to connect to the vhost-user backend, and
cfg(windows) uses a Tube.

BUG=b:249361790
TEST=tools/presubmit --all

Change-Id: I47651060c2ce3a7e9f850b7ed9af8bd035f82de6
```

- The first line is a subject that starts with a tag that represents which components your commit relates to. Tags are usually the name of the crate you modified such as `devices:` or `base:`. If you only modified a specific component in a crate, you can specify the path to the component as a tag like `devices: vhost: user:`. If your commit modified multiple crates, specify the crate where your main change exists. The subject should be no more than 50 characters, including any tags.
- The body should consist of a motivation followed by an impact/action. The body text should be wrapped to 72 characters.
- `BUG` lines are used to specify an associated issue number. If the issue is filed at Google's issue tracker, write `BUG=b:<bug number>`. If no issue is associated, write `BUG=None`. You can have multiple `BUG` lines.
- `TEST` lines are used to describe how you tested your commit in a free form. You can have multiple `TEST` lines.
- `Change-Id` is used to identify your change on Gerrit. It's inserted by the gerrit commit message hook as explained in the previous section. If a new commit is uploaded with the same `Change-Id` as an existing CL's `Change-Id`, gerrit will recognize the new commit as a new patchset of the existing CL.

Uploading changes

To make changes to crosvm, start your work on a new branch tracking `origin/main`.

```
git checkout --branch myfeature --track origin/main
```

After making the necessary changes, and testing them via Presubmit Checks, you can commit and upload them:

```
git commit
./tools/cl upload
```

If you need to revise your change, you can amend the existing commit and upload again:

```
git commit --amend
./tools/cl upload
```

This will create a new version of the same change in gerrit.

> Note: We don't accept any pull requests on the GitHub mirror.

Getting Reviews

All submissions needs to be reviewed by one of the crosvm owners. Use the gerrit UI to request a review. If you are uncertain about the correct person to review, reach out to the team via chat or email list.

Submitting code

Crosvm uses a Commit Queue, which will run pre-submit testing on all changes before merging them into crosvm.

Once one of the crosvm owners has voted "Code-Review+2" on your change, you can use the "Submit to CQ" button, which will trigger the test process.

Gerrit will show any test failures. Refer to Building Crosvm for information on how to run the same tests locally.

When all tests pass, your change is merged into `origin/main`.

Contributing to the documentation

The book of crosvm is built with mdBook. Each markdown file must follow Google Markdown style guide.

To render the book locally, you need to install mdbook and mdbook-mermaid, which should be installed when you run `./tools/install-deps` script. Or you can use the `tools/dev_container` environment.

```
cd docs/book/
mdbook build
```

Output is found at `docs/book/book/html/`.

> Note: If you make a certain size of changes, it's recommended to reinstall mdbook manually with `cargo install mdbook`, as `./tools/install-deps` only installs a binary with some convenient features disabled. For example, the full version of mdbook allows you to edit files while checking rendered results.

Style guide for platform specific code

Code organization

The crosvm code can heavily interleave platform specific code into platform agnostic code using `#[cfg(target_os = "")]`. This is difficult to maintain as

- It reduces readability.
- Difficult to write/maintain unit tests.
- Difficult to maintain downstream, proprietary code

To address the above mentioned issue, the style guide provides a way to standardize platform specific code layout.

Consider a following example where we have platform independent code, `PrintInner`, which is used by platform specific code, `WinPrinter` and `UnixPrinter` to tweak the behavior according to the underlying platform. The users of this module, `sys`, get to use an aliased struct called `Printer` which exports similar interfaces on both the platforms.

In this scheme `print.rs` contains platform agnostic logic, structures and traits. Different platforms, in `unix.rs` and `windows.rs`, implement traits defined in `print.rs`. Finally `sys.rs` exports interfaces implemented by platform specific code.

In a more complex library, we may need another layer, `print.rs`, that uses traits and structures exported by platform specific code, `unix/print.rs` and `windows/print.rs`, and adds some more common logic to it. Following example illustrates the scheme discussed above. Here, `Printer.print()` is supposed to print a value of `u32` and print the target os name.

The files that contain platform specific code **only** should live in a directory named `sys/` and those files should be conditionally imported in `sys.rs` file. In such a setup, the directory structure would look like,

```
$  tree
.
├── print.rs
├── sys
│   ├── unix
│   │   └── print.rs
│   ├── unix.rs
│   ├── windows
│   │   └── print.rs
│   └── windows.rs
└── sys.rs
```

File: `print.rs`

```rust
pub struct PrintInner {
    pub value: u32,
}

impl PrintInner {
    pub fn new(value: u32) -> Self {
        Self { value }
    }

    pub fn print(&self) {
        print!("My value:{} ", self.value);
    }
}

// This is useful if you want to
// * Enforce interface consistency or
// * Have more than one compiled-in struct to provide the same api.
//   Say a generic gpu driver and high performance proprietary driver
//   to coexist in the same namespace.
pub trait Print {
    fn print(&self);
}
```

File: `sys/windows/print.rs`

```rust
use crate::print::{Print, PrintInner};

pub struct WinPrinter {
    inner: PrintInner,
}

impl WinPrinter {
    pub fn new(value: u32) -> Self {
        Self {
            inner: PrintInner::new(value),
        }
    }
}

impl Print for WinPrinter {
    fn print(&self) {
        self.inner.print();
        println!("from win");
    }
}
```

File: `sys/unix/print.rs`

```rust
use crate::print::{Print, PrintInner};

pub struct UnixPrinter {
    inner: PrintInner,
}

impl UnixPrinter {
    pub fn new(value: u32) -> Self {
        Self {
            inner: PrintInner::new(value),
        }
    }
}

impl Print for UnixPrinter {
    fn print(&self) {
        self.inner.print();
        println!("from unix");
    }
}
```

File: `sys.rs`

```rust
cfg_if::cfg_if! {
    if #[cfg(unix)] {
        mod unix;
        pub use platform_print::UnixPrinter as Printer;
    } else if #[cfg(windows)] {
        mod windows;
        pub use platform_print::WinPrinter as Printer;
    }
}
```

Imports

When conditionally importing and using modules, use `cfg(unix)` and `cfg(windows)` for describing the platform. Order imports such that common comes first followed by unix and windows dependencies.

```
// All other imports

#[cfg(unix)]
use {
  std::x::y,
  base::a::b::{Foo, Bar},
  etc::Etc,
};

#[cfg(windows)]
use {
  std::d::b,
  base::f::{Foo, Bar},
  etc::{WinEtc as Etc},
};
```

Structure

It is OK to have a few platform specific fields inlined with cfgs. When inlining

- Ensure that all the fields of a particular platform are next to each other.
- Organize common fields first and then platform specific fields ordered by the target os name i.e. "unix" first and "windows" later.

If the structure has a large set of fields that are platform specific, it is more readable to split it into different platform specific structures and have their implementations separate. If necessary, consider defining a crate in platform independent and have the platform specific files implement parts of those traits.

Enum

When enums need to have platform specific variants

- Create a new platform specific enum and move all platform specific variants under the new enum
- Introduce a new variant, which takes a platform specific enum as member, to platform independent enum.

Do

File: `sys/unix/base.rs`

```
enum MyEnumSys {
  Unix1,
}

fn handle_my_enum_impl(e: MyEnumSys) {
  match e {
    Unix1 => {..},
  };
}
```

File: `sys/windows/base.rs`

```
enum MyEnumSys {
  Windows1,
}

fn handle_my_enum_impl(e: MyEnumSys) {
  match e {
    Windows1 => {..},
  };
}
```

File: `base.rs`

```
use sys::MyEnumSys;
enum MyEnum {
  Common1,
  Common2,
  SysVariants(MyEnumSys),
}

fn handle_my_enum(e: MyEnum) {
  match e {
    Common1 => {..},
    Common2 => {..},
    SysVariants(v) => handle_my_enum_impl(v),
  };
}
```

Don't

File: `base.rs`

```rust
enum MyEnum {
  Common1,
  Common2,
  #[cfg(target_os = "windows")]
  Windows1, // We shouldn't have platform-specific variants in a platform-independent
enum.
  #[cfg(target_os = "unix")]
  Unix1, // We shouldn't have platform-specific variants in a platform-independent
enum.
}

fn handle_my_enum(e: MyEnum) {
  match e {
    Common1 => {..},
    Common2 => {..},
    #[cfg(target_os = "windows")]
    Windows1 => {..}, // We shouldn't have platform-specific match arms in a platform-
independent code.
    #[cfg(target_os = "unix")]
    Unix1 => {..}, // We shouldn't have platform-specific match arms in a platform-
independent code.
  };
}
```

Code blocks and functions

If a code block or a function has little platform independent code and the bulk of the code is platform specific then carve out platform specific code into a function. If the carved out function does most of what the original function was doing and there is no better name for the new function then the new function can be named by appending `_impl` to the functions name.

Do

File: `base.rs`

```rust
fn my_func() {
  print!("Hello ");
  my_func_impl();
}
```

File: `sys/unix/base.rs`

```rust
fn my_func_impl() {
  println!("unix");
}
```

File: `sys/windows/base.rs`

```
fn my_func_impl() {
  println!("windows");
}
```

Don't

File: `base.rs`

```
fn my_func() {
  print!("Hello ");

  #[cfg(target_os = "unix")] {
    println!("unix"); // We shouldn't have platform-specific code in a platform-
independent code block.
  }

  #[cfg(target_os = "windows")] {
    println!("windows"); // We shouldn't have platform-specific code in a platform-
independent code block.
  }
}
```

match

With an exception to matching enums, see enum, matching for platform specific values can be done in the wildcard patter(`_`) arm of the match statement.

Do

File: `parse.rs`

```
fn parse_args(arg: &str) -> Result<()>{
  match arg {
    "path" => {
      <multiple lines of logic>;
      Ok(())
    },
    _ => parse_args_impl(arg),
  }
}
```

File: `sys/unix/parse.rs`

```
fn parse_args_impl(arg: &str) -> Result<()>{
  match arg {
    "fd" => {
      <multiple lines of logic>;
      Ok(())
    },
    _ => Err(ParseError),
  }
}
```

File: `sys/windows/parse.rs`

```
fn parse_args_impl(arg: &str) -> Result<()>{
  match arg {
    "handle" => {
      <multiple lines of logic>;
      Ok(())
    },
    _ => Err(ParseError),
  }
}
```

Don't

File: `parse.rs`

```
fn parse_args(arg: &str) -> Result<()>{
  match arg {
    "path" => Ok(()),
    #[cfg(target_os = "unix")]
    "fd" => { // We shouldn't have platform-specific match arms in a platform-
independent code.
      <multiple lines of logic>;
      Ok(())
    },
    #[cfg(target_os = "windows")]
    "handle" => { // We shouldn't have platform-specific match arms in a platform-
independent code.
      <multiple lines of logic>;
      Ok(())
    },
    _ => Err(ParseError),
  }
}
```

Errors

Inlining all platform specific error values is ok. This is an exception to the enum to keep error handling simple. Organize platform independent errors first and then platform specific errors ordered by the target os name i.e. "unix" first and "windows" later.

Platform specific symbols

If a platform exports symbols that are specific to the platform only and are not exported by all other platforms then those symbols should be made public through a namespace that reflects the name of the platform.

File: `sys.rs`

```
cfg_if::cfg_if! {
    if #[cfg(unix)] {
        pub mod unix;
        use unix as platform;
    } else if #[cfg(windows)] {
        pub mod windows;
        use windows as platform;
    }
}

pub use platform::print;
```

File: `unix.rs`

```
fn print() {
  println!("Hello unix");
}

fn print_u8(val: u8) {
  println!("Unix u8:{}", val);

}
```

File: `windows.rs`

```
fn print() {
  println!("Hello windows");
}

fn print_u16(val: u16) {
  println!("Windows u16:{}", val);

}
```

The user of the library, say mylib, now has to do something like below which makes it explicit that the functions `print_u8` and `print_u16` are platform specific.

```rust
use mylib::sys::print;

fn my_print() {
  print();

  #[cfg(unix)]
  mylib::sys::unix::print_u8(1);

  #[cfg(windows)]
  mylib::sys::windows::print_u16(1);
}
```

Onboarding Resources

Various links to useful resources for learning about virtual machines and the technology behind crosvm.

Talks

Chrome University by zachr (2018, 30m)

- Life of a Crostini VM (user click -> terminal opens)
- All those French daemons (Concierge, Maitred, Garcon, Sommelier)

NYULG: Crostini by zachr / reveman (2018, 50m)

- Overlaps Chrome University talk
- More details on wayland / sommelier from reveman
- More details on crostini integration of app icons, files, clipboard
- Lots of demos

Introductory Resources

OS Basics

- OSDev Wiki (A lot of articles on OS development)
- PCI Enumeration (Most of our devices are on PCI, this is how they are found)
- ACPI Source Language Tutorial

Rust

- Rust Cheat Sheet Beautiful website with idiomatic rust examples, overview of pointer- and container types
- Rust Programming Tipz (with a z, that's how you know it's cool!)
- Rust design patterns repo
- Organized collection of blog posts on various Rust topics

KVM Virtualization

- Low-level tutorial on how to run code via KVM
- KVM Hello World sample program (host + guest)
- KVM API docs
- Awesome Virtualization (Definitely check out the Hypervisor Development section)

Virtio (device emulation)

- Good overview of virtio architecture from IBM
- Virtio drivers overview by RedHat
- Virtio specs (so exciting, I can't stop reading)
- Basics of devices in QEMU

VFIO (Device passthrough)

- Introduction to PCI Device Assignment with VFIO

Virtualization History and Basics

- By the end of this section you should be able to answer the following questions
  - What problems do VMs solve?
  - What is trap-and-emulate?
  - Why was the x86 instruction set not "virtualizable" with just trap-and-emulate?
  - What is binary translation? Why is it required?
  - What is a hypervisor? What is a VMM? What is the difference? (If any)
  - What problem does paravirtualization solve?
  - What is the virtualization model we use with Crostini?
  - What is our hypervisor?
  - What is our VMM?
- CMU slides go over motivation, why x86 instruction set wasn't "virtualizable" and the good old trap-and-emulate
- Why Intel VMX was needed; what does it do (Link)
- What is a VMM and what does it do (Link)
- Building a super simple VMM blog article (Link)

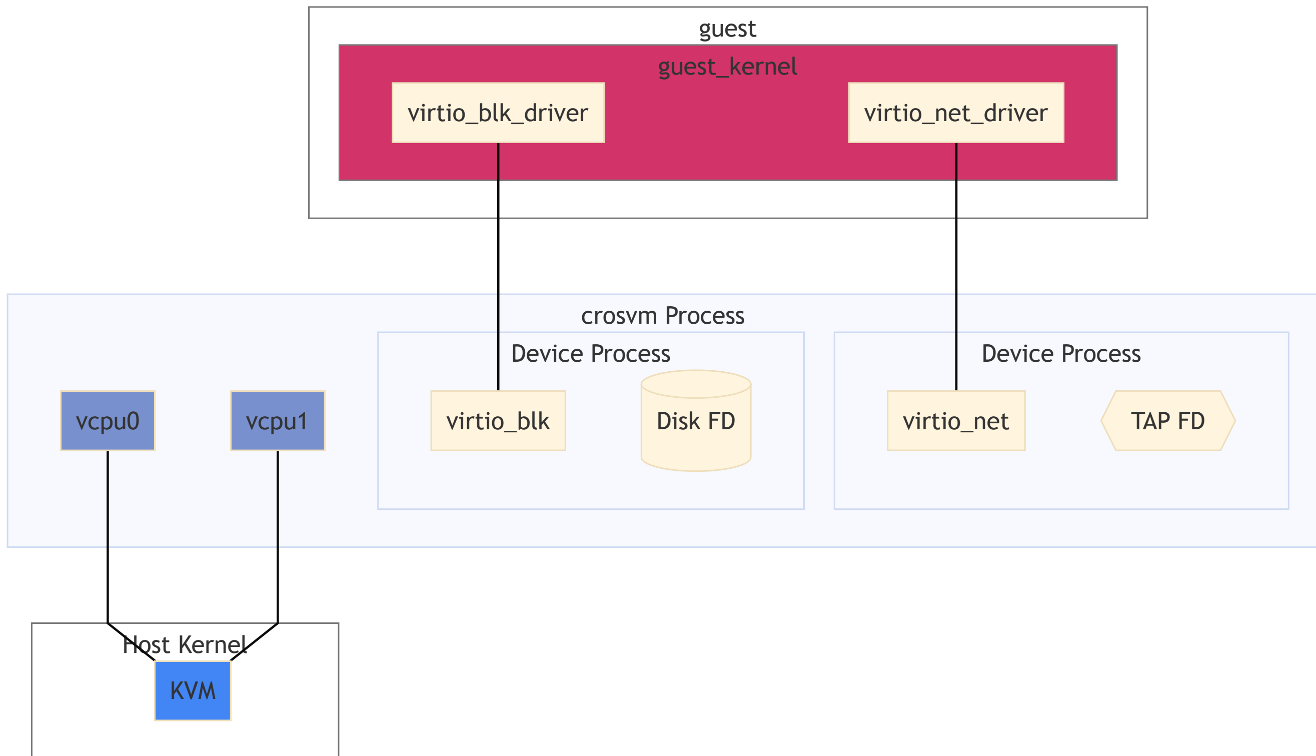
Relevant Specs

- ACPI Specs
- DeviceTree Specs
- Vhost-user protocol

Appendix

The following sections contain reference material you may find useful when working on crosvm. Note that some of contents might be outdated.

Sandboxing



Generally speaking, sandboxing is achieved in crosvm by isolating each virtualized devices into its own process. A process is always somewhat isolated from another by virtue of being in a different address space. Depending on the operating system, crosvm will use additional measures to sandbox the child processes of crosvm by limiting each process to just what it needs to function.

In the example diagram above, the virtio block device exists as a child process of crosvm. It has been limited to having just the FD needed to access the backing file on the host and has no ability to open new files. A similar setup exists for other devices like virtio net.

Seccomp

The seccomp system is used to filter the syscalls that sandboxed processes can use. The form of seccomp used by crosvm (`SECCOMP_SET_MODE_FILTER`) allows for a BPF program to be used. To generate the BPF programs, crosvm uses minijail's policy file format. A policy file is written for each device per architecture. Each device requires a unique set of syscalls to accomplish their function and each architecture has slightly different naming for similar syscalls. The ChromeOS docs have a useful listing of syscalls.

Writing a Policy for crosvm

The detailed rules for naming policy files can be found in seccomp/README.md

Most policy files will include the `common_device.policy` from a given architecture using this directive near the top:

```
@include /usr/share/policy/crosvm/common_device.policy
```

The common device policy for `x86_64` is:

```
@frequency ./common_device.frequency
brk: 1
clock_gettime: 1
clone: arg0 & CLONE_THREAD
clone3: 1
close: 1
dup2: 1
dup: 1
epoll_create1: 1
epoll_ctl: 1
epoll_pwait: 1
epoll_wait: 1
eventfd2: 1
exit: 1
exit_group: 1
futex: 1
getcwd: 1
getpid: 1
gettid: 1
gettimeofday: 1
io_uring_setup: 1
io_uring_register: 1
io_uring_enter: 1
kill: 1
lseek: 1
madvise: arg2 == MADV_DONTNEED || arg2 == MADV_DONTDUMP || arg2 == MADV_REMOVE || arg2
== MADV_MERGEABLE || arg2 == MADV_FREE
membarrier: 1
mmap: arg2 in ~PROT_EXEC
mprotect: arg2 in ~PROT_EXEC
mremap: 1
munmap: 1
nanosleep: 1
clock_nanosleep: 1
pipe2: 1
poll: 1
ppoll: 1
read: 1
readlink: 1
readlinkat: 1
readv: 1
recvfrom: 1
recvmsg: 1
restart_syscall: 1
rseq: 1
rt_sigaction: 1
rt_sigprocmask: 1
rt_sigreturn: 1
sched_getaffinity: 1
sched_yield: 1
sendmsg: 1
sendto: 1
set_robust_list: 1
sigaltstack: 1
write: 1
writev: 1
fcntl: 1
uname: 1

## Rules for vmm-swap
```

```
userfaultfd: 1
# 0xc018aa3f == UFFDIO_API, 0xaa00 == USERFAULTFD_IOC_NEW
ioctl: arg1 == 0xc018aa3f || arg1 == 0xaa00
```

The syntax is simple: one syscall per line, followed by a colon `:`, followed by a boolean expression used to constrain the arguments of the syscall. The simplest expression is `1` which unconditionally allows the syscall. Only simple expressions work, often to allow or deny specific flags. A major limitation is that checking the contents of pointers isn't possible using minijail's policy format. If a syscall is not listed in a policy file, it is not allowed.

Memory Layout

x86-64 guest physical memory map

This is a survey of the existing memory layout for crosvm on x86-64 when booting a Linux kernel. Some of these values are different when booting a BIOS image or when compiled with features=direct (ManaTEE); see the source. All addresses are in hexadecimal.

Name/source link	Address	End (exclusive)	Size	Notes
	0000	7000		RAM (may start at 0x1000 for crosvm-direct)
ZERO_PAGE_OFFSET	7000			Linux boot_params structure
BOOT_STACK_POINTER	8000			Boot SP value
boot_pml4_addr	9000			Boot page table
boot_pdpte_addr	A000			Boot page table
boot_pde_addr	B000			Boot page table
CMDLINE_OFFSET	2_0000	20_0000	~1.87 MiB	Linux kernel command line
ACPI_HI_RSDP_WINDOW_BASE	E_0000			ACPI RSDP table (TODO: technically overlaps command line buffer; check CMDLINE_MAX_SIZE
KERNEL_START_OFFSET	20_0000			Linux kernel image load address
END_ADDR_BEFORE_32BITS	20_0000	D000_0000	~3.24 GiB	RAM (<4G)
END_ADDR_BEFORE_32BITS	D000_0000	F400_0000	576 MiB	Low (<4G) MMIO allocation area
PCIE_CFG_MMIO_START	F400_0000	F800_0000	64 MiB	PCIe enhanced conf (ECAM)
RESERVED_MEM_SIZE	F800_0000	1_0000_0000	128 MiB	LAPIC/IOAPIC/HPET/
TSS_ADDR	FFFB_D000			Boot task state segment
	1_0000_0000			RAM (>4G)
	(end of RAM)			High (>4G) MMIO allocation area

aarch64 guest physical memory map

All addresses are IPA in hexadecimal.

Common layout

These apply for all boot modes.

Name/source link	Address	End (exclusive)	Size	Notes
`SERIAL_ADDR[3]`	`2e8`	`2f0`	8 bytes	Serial port MMIO
`SERIAL_ADDR[1]`	`2f8`	`300`	8 bytes	Serial port MMIO
`SERIAL_ADDR[2]`	`3e8`	`3f0`	8 bytes	Serial port MMIO
`SERIAL_ADDR[0]`	`3f8`	`400`	8 bytes	Serial port MMIO
`AARCH64_RTC_ADDR`	`2000`	`3000`	4 KiB	Real-time clock
`AARCH64_VMWDT_ADDR`	`3000`	`4000`	4 KiB	Watchdog device
`AARCH64_PCI_CFG_BASE`	`1_0000`	`2_0000`	64 KiB	PCI configuration (CAM)
`AARCH64_PVTIME_IPA_START`	`1f0_0000`	`200_0000`	64 KiB	Paravirtualized time
`AARCH64_MMIO_BASE`	`200_0000`	`400_0000`	32 MiB	Low MMIO allocation area
`AARCH64_GIC_CPUI_BASE`	`3ffd_0000`	`3fff_0000`	128 KiB	vGIC
`AARCH64_GIC_DIST_BASE`	`3fff_0000`	`4000_0000`	64 KiB	vGIC
`AARCH64_AXI_BASE`	`4000_0000`			Seemingly unused? Is this hard-coded somewhere in the kernel?
`AARCH64_PROTECTED_VM_FW_START`	`7fc0_0000`	`8000_0000`	4 MiB	pVM firmware (if running a protected VM)
`AARCH64_PHYS_MEM_START`	`8000_0000`		--mem size	RAM (starts at IPA = 2 GiB)
`get_swiotlb_addr`	after RAM		--swiotlb	Only present for

Name/source link	Address	End (exclusive)	Size	Notes
			size	hypervisors requiring static swiotlb alloc
plat_mmio_base	after swiotlb	+0x800000	8 MiB	Platform device MMIO region
high_mmio_base	after plat_mmio	max phys addr		High MMIO allocation area

Layout when booting a kernel

These apply when no bootloader is passed, so crosvm boots a kernel directly.

Name/source link	Address	End (exclusive)	Size	Notes
AARCH64_KERNEL_OFFSET	8000_0000			Kernel load location in RAM
initrd_addr	after kernel			Linux initrd location in RAM
fdt_address	before end of RAM		2 MiB	Flattened device tree in RAM

Layout when booting a bootloader

These apply when a bootloader is passed with `--bios`.

Name/source link	Address	End (exclusive)	Size	Notes
AARCH64_FDT_OFFSET_IN_BIOS_MODE	8000_0000	8020_0000	2 MiB	Flattened device tree in RAM
AARCH64_BIOS_OFFSET	8020_0000			Bootloader image in RAM

Minijail

On Linux hosts, crosvm uses minijail to sandbox the child devices. The minijail C library is utilized via a Rust wrapper so as not to repeat the intricate sequence of syscalls used to make a secure isolated child process. The fact that minijail was written, maintained, and continuously tested by a professional security team more than makes up for its being written in an memory unsafe language.

The exact configuration of the sandbox varies by device, but they are mostly alike. See `create_base_minijail` from `linux/jail_helpers.rs`. The set of security constraints explicitly used in crosvm are:

- PID Namespace
  - Runs as init
- Deny setgroups
- Optional limit the capabilities mask to `0`
- User namespace
  - Optional uid/gid mapping
- Mount namespace
  - Optional pivot into a new root
- Network namespace
- PR_SET_NO_NEW_PRIVS
- seccomp with optional log failure mode
- Limit to number of file descriptors

Package Documentation

The package documentation generated by `cargo doc` is available here.