The `build2` Operating System

Revision `0.18`, June 2024
This revision of the document describes the `build2` Build OS `0.18.x` series.

Table of Contents

Preface

This document describes Build OS, the `build2` build operating system.

1 Introduction

Build OS is a Debian GNU/Linux-based in-memory network-booted operating system specialized for autonomous building of software using the `build2` toolchain. It's primary purpose is to run the `build2` build bot (`bbot`), build slave (`bslave`), or both.

A machine that run an instance of Build OS is called a *build host*. A build host runs the `bbot` and/or `bslave` in the *agent mode*. The actual building is performed in the virtual machines and/or containers. For `bbot` it is normally one-shot virtual machines and for `bslave` it is normally containers but can also be long-running virtual machines. Inside virtual machines/containers, `bbot` and `bslave` run in the *worker mode* and receive *build tasks* from their respective agents.

2 Architecture

Build OS root filesystem (`rootfs`) resides entirely in RAM with all changes (such as installation of the `build2` toolchain) discarded on the next reboot. A small amount of persistent (but not precious) state is stored in `/state` (see State). A minimum of 4GB of RAM is required for Build OS itself (that is, excluding any virtual machines and containers).

After booting the kernel, the Build OS execution starts with a custom `init` script which performs low-level configuration and setup and then hands off the initialization to `systemd`. At the end of `systemd` startup the Build OS monitor script (`buildos`) is started as a `systemd` service. On startup this script bootstraps the `build2` toolchain, builds the `bbot` package, and starts it (as another `systemd` service) in the agent mode. After that the monitor starts monitoring for OS and toolchain changes. If either is detected, the host is rebooted, which will trigger both booting the latest OS and building of the latest `build2` toolchain and `bbot`.

@@ TODO: init steps.

The monitor service (and `bbot` that it starts) are executed as the `build` user/group. The home directory of this user is `/build/`. It has the following subdirectories:

- `/build/tftp/`

  A size-limited `tmpfs` filesystem that is used to communicate with build machines as well as for build host log access.

This directory is read-accessible via the TFTP server running on the default port. A `bbot` agent also makes the `bootstrap/` and `build/` sub-directories inside this directory temporarily write-accessible to build machines by running custom instances of the TFTP server on other ports.

- `/build/toolchains/`

  Contains `build2` toolchain installations after bootstrap. Each version is installed into a subdirectory named as the toolchain name. See Toolchains for details.

- `/build/bots/`

  Contains `bbot` installations. Each version is installed into a subdirectory named as the toolchain name. See Toolchains for details.

- `/build/machines/`

  Contains virtual machines and containers. See Machines for details.

3 Booting

Build OS is normally booted over the network using PXE but can also be booted locally from the kernel image and initrd directly.

3.1 Reboot

Build OS can detect when the OS or toolchain have been updated and automatically reboot the build host. This is achieved by polling the URLs specified with the `buildos.buildid_url` and `buildos.toolchain_url` kernel command line parameters.

The `buildos.buildid_url` value should point to the `buildos-buildid` file that comes along the kernel image and initrd. The `buildos.toolchain_url` value is the location of the toolchain checksums file as described in Toolchains. See Network Boot for the usage example.

3.2 Network Boot

Here we assume that you have already established your PXE setup using PXELINUX. That is, you have configured a TFTP server that hosts the `pxelinux` initial bootstrap program (NBP) and configured a DHCP server to direct PXE client to this server/NBP.

To setup PXE boot of Build OS, perform the following steps:

1. Copy the Build OS `-image`, `-initrd`, and `-buildid` files to the TFTP server. For example:

```
# mkdir -p /var/lib/tftpboot/buildos
# cp buildos-image buildos-initrd buildos-buildid \
  /var/lib/tftpboot/buildos/
```

2. Assuming the host MAC address is `de:ad:be:ef:b8:da`, create a host-specific configuration file (or use `default` as the last path component for a configuration that applies to all hosts):

```
# cat <<EOF >/var/lib/tftpboot/pxelinux.cfg/01-de-ad-be-ef-b8-da
default buildos
prompt 1
timeout 50

label buildos
  menu label Build OS
  kernel /buildos/buildos-image
  initrd /buildos/buildos-initrd
  append buildos.smtp_relay=example.org buildos.admin_email=admin@example.org buildos.buildid_url=tftp://<os-host>/buildos/buildos-buildid buildos.toolchain_url=https://<toolchain-host>/toolchain.sha256 buildos.toolchain_trust=<repo-fp>
EOF
```

   Where `<os-host>` is the address of the TFTP server (the same address as returned by the DHCP server to PXE clients), `<toolchain-host>` is the host that serves the toolchain archives, and `<repo-fp>` is the toolchain repository certificate fingerprint to trust. Note that all the parameters in `append` must be specified on a single line.

3. You can test the setup using QEMU/KVM, for example:

```
$ sudo kvm \
  -m 8G \
  -netdev tap,id=net0,script=./qemu-ifup \
  -device e1000,netdev=net0,mac=de:ad:be:ef:b8:da \
  -boot n
```

3.3 Local Boot

During testing it is often useful to boot Build OS directly from the kernel image and initrd files. As an example, here is how this can be done using QEMU/KVM:

```
sudo kvm \
  -m 8G \
  -netdev tap,id=net0,script=./qemu-ifup \
  -device e1000,netdev=net0,mac=de:ad:be:ef:b8:da \
  -kernel buildos-image -initrd buildos-initrd
```

4 Configuration

4.1 CPU and RAM

A Build OS instances divides available CPUs and RAM (minus reserved, see below) into *slices* that are then *committed* to each instance of each toolchain. In case of CPU it normally makes sense to overcommit this resource in order to improve utilization. This can be achieved by specifying the overcommit values as a ratio with `buildos.cpu_overcommit`. For example, given

the following CPU overcommit:

```
buildos.cpu_overcommit=3/2
```

A Build OS machine with 8 CPUs (hardware threads) and three instances will assign 4 CPUs (`8 * 3/2 / 3`) to each slice.

It is also possible to specify CPU affinity with `buildos.cpu_affinity`. For example, specifying:

```
buildos.cpu_affinity=2-9
```

Will restrict the instances to only running on CPUs 2-9.

It is possible to reserve a number of CPUs and an amount of RAM to Build OS with the `buildos.cpu_reserved` and `buildos.ram_reserved` (in GiB) kernel command line parameters. If unspecified, 4GiB of RAM is reserved by default.

An amount of RAM can be reserved for auxiliary machines with `buildos.ram_auxiliary`. This amount will also be divided into slices and committed to each instance.

Finally, if the total available RAM cannot be auto-detected, it can be specified manually with `buildos.ram_total`. Here is a complete example of specifying all the possible RAM values:

```
buildos.ram_total=64
buildos.ram_reserved=4
buildos.ram_auxiliary=12
```

Assuming three instances, the configuration will assign 16GiB of build and 4GiB of auxiliary RAM to each instance and keep 4GiB reserved to Build OS.

4.2 Storage

Build OS configures storage based on the labels assigned to disks and partitions (collectively refered to as disks from now on). Build OS requires storage for state as well as virtual machines and containers.

4.2.1 State

Build OS stores a small amount of state on a disk labeled `buildos.state` (mounted as `/state`). This includes random number generator state, SSH server host keys, and so on. While this state is persistent, it is not precious.

The stored state is fairly small (hundreds of megabytes) and is not performance-critical. While one can create a small state partition on the same physical disk as used for machines (see below), having it on a separate disk makes it easier to move machine disks around. Based on these requirements, a small, good-quality USB flash drive or flash card is a good option.

While any suitable filesystem can be used, `ext4` is a good choice, with journaling disabled if used on a flash drive/card. For example:

```
mkfs.ext4 -L buildos.state -O ^has_journal /dev/sdX
```

Flash drives and cards tend to fail over time and while the state is not precious, recreating it would also require updating the public key on all the controllers that this build host serves. As a result, it may be prudent to backup it up.

4.2.2 Machines

For virtual machine and container storage we can use a single disk, in which case it can be labeled just `buildos.machines`. If we would like to use multiple disks, then they should be labeled `buildos.machines.<volume>`. In both cases the disks must be formatted as `btrfs`.

In a single disk configuration, the disk is mounted as `/build/machines/default/` (in other words, as the default valume called `default`). In a multi-disk configuration, each disk is mounted as `/build/machines/<volume>/`.

If no disks are found for required storage, then the boot process is interrupted with a shell prompt where you can format and/or lable a suitable disk. You can also view the storage configuration on a booted Build OS instance by examining `/etc/fstab`.

As an example, let's consider the first boot of a clean machine that has an SSD disk as `/dev/sda` and which we would like to use for virtual machine storage. We would also like to over-provision this SSD by 10% to (potentially) prolong its life and increase performance (you may want to skip this step if you are using a datacenter-grade SSD that would normally already be generously over-provisioned).

On the first boot we will be presented with a shell prompt which we use to over-provision the disk:

```
# fdisk -l /dev/sda             # Query disk information.
# hdparm -N /dev/sda            # Query disk/host protection area sizes.
# hdparm -Np<COUNT> /dev/sda    # COUNT = sector count * 0.9
# hdparm -N /dev/sda            # Verify disk/host protection area sizes.
# ^D                            # Exit shell and reboot.
```

Note that this may not always work, depending on the disk controller used. An alternative approach is to use the `mkfs.btrfs --byte-count` option when formatting the disk to leave some disk space untouched and unused.

After the reboot we will be presented with a shell prompt again where we confirm over-provisioning, format the disk as `btrfs`, and label it as `buildos.machines`:

```
# fdisk -l /dev/sda            # Confirm disk size decreased by 10%.
# mkfs.btrfs -L buildos.machines -m single /dev/sda
# ^D                           # Exit shell and reboot.
```

To create a single `btrfs` disk that spans multiple physical devices:

```
# mkfs.btrfs -L buildos.machines -d single -m single /dev/sda /dev/sdb
```

4.3 Network

Network is configured via DHCP. Initially, all Ethernet interfaces that have carrier are tried in (some) order and the first interface that is successfully configured via DHCP is used.

Hostname is configured from the DHCP information. Failed that, a name is generated based on the MAC address, in the form `build-xxxxxxxxxx`. @@ Maybe also kernel cmdline?

Based on the discovery of the Ethernet interface, two bridge interfaces are configured: `br0` is a public bridge that includes the Ethernet interface and is configured via DHCP. `br1` is a private interface with NAT to `br0` with `dnsmasq` configured as a DHCP on this interface.

Normally, `br0` is used for `bslave` virtual machines/container (since they may need to be accessed directly) and `br1` – for `bbot` virtual machines. You can view the bridge configuration on a booted Build OS instance by examining `/etc/network/interfaces`.

@@ TODO: private network parameters.

4.4 Email

A Build OS instance sends various notifications (including all messages to `root`) to the admin email address. The admin email is specified with the `buildos.admin_email` kernel command line parameter.

In order to deliver mail, the `postfix` MTA is configured to forward to a relay. The relay host is specified with the `buildos.smtp_relay` kernel command line parameter.

Note that no authentication of any kind is configured for relaying. This means that the relay host should accept emails from build hosts either because of their network location (for example, because they are on your organization's local network and you are using your organization's

relay) or because the relay host accepts emails send to the admin address from anyone (which is normally the case if the relay is the final destination for the admin address, for example, `example.org` and `admin@example.org`).

4.5 SSH

Build OS runs an OpenSSH server with password authentication and `root` login disabled. As a result, the only way to login remotely is as user `build` using public key authentication. To add a public key into the `build`'s `authorized_keys` file we can use the `buildos.ssh_key` kernel command line parameter. For example (note the quotes):

```
buildos.ssh_key="ssh-rsa AAA...OA0DB user@host"
```

4.6 Toolchains

The first step performed by the Build OS monitor is to bootstrap the `build2` toolchain. The location of the toolchain packages is specified with the `buildos.toolchain_url` kernel command line parameter. This URL should point to the *toolchain checksums file*. You will also normally need to pass the `buildos.toolchain_trust` parameter which is the toolchain repository certificate fingerprint that the monitor should trust. Note also that the bootstrap process (both on the build host and inside build machines) uses the default toolchain repository location embedded into the build scripts in the `build2-toolchain` package.

It is also possible to use multiple toolchains on a single Build OS instance. In this case a toolchain name can be appended after `buildos.toolchain_*`, for example, `buildos.toolchain_url.<name>` (values without the toolchain name use the toolchain name `default`). The toolchain name may not contain `-`.

Each toolchain may also execute multiple `bbot` agent instances. The number of instances is specified with the `buildos.instances[.<name>]` parameter.

All `bbot` agent instances of a toolchain are executed with the same nice value which can be specified with the `buildos.nice[.<name>]` parameter. It should be between -20 (highest priority) and 19 (lowest priority) with 0 being the default. See **`sched(7)`** for details.

The bridge interface to be used for machine networking can be specified with the `buildos.bridge[.<name>]` parameter. Valid values are `br0` (public bridge to the physical interface) and `br1` (private/NAT'ed bridge to `br0`). If unspecified, `br1` is used by default.

In the checksums file blank lines and lines that start with `#` are ignored. If the first line is the special `disabled` value, then this toolchain is ignored. Otherwise, each line in the checksums file is the output of the `shaNNNsum(1)` utility, that is, the SHANNN sum following by space, an asterisk (`*`, which signals the binary mode), and the relative file path. The extension of the

checksums file should be `.shaNNN` and the first line should be for the `build2-toolchain` `tar` archive itself (used to derive the toolchain version). For example:

```
# toolchain.sha256
ae89[...]87a4 *0.4.0/build2-toolchain-0.4.0.tar.xz
058d[...]c962 *0.4.0/build2-baseutils-0.4.0-x86_64-windows.zip
e723[...]c305 *0.4.0/build2-mingw-0.4.0-x86_64-windows.tar.xz
```

Based on the checksums file the monitor downloads each file into `/build/tftp/toolchains/<name>/` (the file path is taken as relative to `toolchain_url`), verifies their checksums, and creates *predictable name* symlinks (names without the version). It also creates the `version` which contains the toolchain version and the `trust` file which contains the value of the `buildos.toolchain_trust` parameter or the special `"no"` value if none were specified.

Continuing with the above example, the contents of `/build/tftp/toolchains/default/` would be:

```
version
trust

build2-toolchain-0.4.0.tar.xz
build2-baseutils-0.4.0-x86_64-windows.zip
build2-mingw-0.4.0-x86_64-windows.tar.xz

build2-toolchain-tar.xz             -> build2-toolchain-0.4.0.tar.xz
build2-baseutils-x86_64-windows.zip -> build2-baseutils-0.4.0-x86_64-windows.zip
build2-mingw-x86_64-windows.tar.xz  -> build2-mingw-0.4.0-x86_64-windows.tar.xz
```

While the monitor itself only needs the `build2-toolchain` package, build machine toolchain bootstrap may require additional packages (which will be accessed via TFTP using predictable names).

4.7 Controllers

For each toolchain the `bbot` agent polls one or more controllers for build tasks to perform. The controller URLs are configured with the the `buildos.controller_url[.<name>]` kernel command line parameter (where `<name>` is optional toolchain name). To specify multiple controllers, repeat this parameter.

Additionally, we can use the `buildos.controller_trust[.<name>]` kernel command line parameter to specify SHA256 repository certificate fingerprints to trust (see the `trust` build task manifest value for details). To specify multiple fingerprints, repeat this parameter.

5 Build Machines

At the top level, a machine storage volume (see Machines) contains machine directories, for example:

```
/build/machines/default/
|-- linux-gcc_6/
`-- windows-msvc_14/
```

The layout inside a machine directory is as follows, where `<name>` is the machine name and `<toolchain>` is the toolchain name:

```
<name>/
|-- <name>-1   -> <name>-1.1
|-- <name>-1.0/
|-- <name>-1.1/
|-- <name>-<toolchain>/
`-- <name>-<toolchain>-<xxx>/
```

The `<name>-<P>.<R>` entries are read-only `btrfs` subvolumes that contain the initial (that is, *pre-bootstrap*) machine images. The numeric `<P>` part indicates the *bootstrap protocol version*. The numeric `<R>` part indicates the machine revision.

The `<name>-<P>` entry is a symbolic link to `<name>-<P>.<N>` that is currently in effect.

The `<name>-<toolchain>` entry is the bootstrapped machine image for `<toolchain>`. It is created by cloning `<name>-<P>` with a bootstrap protocol version that matches this toolchain's `bbot` and then bootstrapping the `build2` toolchain inside.

The `<name>-<toolchain>-<xxx>` entries are the temporary snapshots of `<name>-<toolchain>` created by `bbot` for building packages.

A machine can be added, upgraded, or removed on a live Build OS instance. This needs to be done in a particular order to avoid inconsistencies and race conditions.

5.1 Adding a Machine

Let's assume you have a read-only `btrfs` `linux-gcc_6-1.0` subvolume on a development host (we will call it `devel`) that contains the initial version of our virtual machine. We would like to add it to the build host (running Build OS, we will call it `build`) into the default machine volume (`/build/machines/default/`). To achieve this in an atomic way we perform the following steps:

```
# Create the machine directory.
#
build$ mkdir /build/machines/default/linux-gcc_6

# Send the machine subvolume to build host.
#
devel$ sudo btrfs send linux-gcc_6-1.0 | \
  ssh build@build sudo btrfs receive /build/machines/default/linux-gcc_6/

build$ cd /build/machines/default/linux-gcc_6

# Make user build the owner of the machine subvolume.
#
build$ sudo btrfs property set -ts linux-gcc_6-1.0 ro false
build$ sudo chown build:build linux-gcc_6-1.0 linux-gcc_6-1.0/*
build$ btrfs property set -ts linux-gcc_6-1.0 ro true

# Make the subvolume the current machine.
#
build$ ln -s linux-gcc_6-1.0 linux-gcc_6-1
```

The `upload-machine` helper script implements this sequence of steps.

5.2 Upgrading a Machine

Continuing with the example started in the previous section, let's assume we have created `linux-gcc_6-1.1` as a snapshot of `linux-gcc_6-1.0` and have made some modification to the virtual machine (all on the development host). We now would like to switch to this new revision of our machine on the build host. To achieve this in an atomic way we perform the following steps:

```
# Send the new machine subvolume to build host incrementally.
#
devel$ sudo btrfs send -p linux-gcc_6-1.0 linux-gcc_6-1.1 | \
  ssh build@build sudo btrfs receive /build/machines/default/linux-gcc_6/

build$ cd /build/machines/default/linux-gcc_6

# Make user build the owner of the new machine subvolume.
#
build$ sudo btrfs property set -ts linux-gcc_6-1.1 ro false
build$ sudo chown build:build linux-gcc_6-1.1 linux-gcc_6-1.1/*
build$ btrfs property set -ts linux-gcc_6-1.1 ro true

# Switch the current machine atomically.
#
build$ ln -s linux-gcc_6-1.1 new-linux-gcc_6-1
build$ mv -T new-linux-gcc_6-1 linux-gcc_6-1

# Remove the old machine subvolume (optional).
#
build$ btrfs property set -ts linux-gcc_6-1.0 ro false
build$ btrfs subvolume delete linux-gcc_6-1.0
```

The `upload-machine` helper script implements this sequence of steps.

5.3 Remove a Machine

Continuing with the example started in the previous section, let's assume we are no longer interested in the `linux-gcc_6` machine and would like to remove it. This operation is complicated by the possibility of `bbot` instances currently building with this machine.

```
build$ cd /build/machines/default/linux-gcc_6

# Remove the current machine symlink.
#
build$ rm linux-gcc_6-1

# Wait for all the linux-gcc_6-<toolchain>-<xxx> subvolumes
# to disappear.
#
build$ for d in linux-gcc_6-*-*/; do           \
  while [ -d $d ]; do \
    echo "waiting for $d" && \
    sleep 10; \
  done; \
done

# Remove the initial and bootstrapped machine subvolume(s).
#
build$ for d in linux-gcc_6-*/; do     \
  btrfs property set -ts $d ro false && \
  btrfs subvolume delete $d; \
done

# Remove the machine directory.
#
build$ cd ..
build$ rmdir /build/machines/default/linux-gcc_6
```

The `remove-machine` helper script implements this sequence of steps.

Note also that on reboot the Build OS monitor examines and cleans up machine directories of any stray subvolumes. As a result, an alternative approach would be to remove the current machine symlink and reboot the build host.

5.4 Logging into a Machine

A running QEMU/KVM machine (that is, one being bootstrapped or used for building) can be accessed with a VNC client. Clients based on `gtk-vnc`, such as `vinagre`, are known to work reasonably well. For example:

```
ssh -f -L 5901:127.0.0.1:5901 build@build sleep 1 && vinagre 127.0.0.1:5901
```

If the machine has been suspended, it can be resumed using the following command:

```
echo cont | ssh build@build socat - UNIX-CONNECT:/tmp/monitor-<toolchain>-<instance>
```

The `login-machine` helper script implements this sequence of steps.

Other useful QEMU monitor commands are `system_powerdown` and `system_reset`.