Eisfair64 LXC Container für Proxmox

Ich mag eisfair, mit seiner leichten und unkomplizierten Konfiguration und habe mal probiert, ob man eisfair nicht auch als LXC-Container in Proxmox laufen lassen kann. Ein LXC-Container bietet gegenüber einer VM einige Vorteile, hat aber natürlich auch einige Nachteile.

Vorteile:

- Nutzen denselben Kernel wie das Host-System, bieten aber isolierte Benutzerbereiche für Anwendungen.
- Schneller einzurichten und verbrauchen weniger Ressourcen im Vergleich zu VMs.
- Schnelleres Starten: Starten viel schneller als VMs, da kein eigener Kernel und Bootloader geladen wird
- Einfachheit: Einfacher einzurichten und zu verwalten als VMs.
- …

Nachteile:

- Beschränkt auf Linux-Systeme
- Schwierigere Integration von Speicherlösungen wie NFS, etc.
- …

Ich wollte das aber einfach mal probieren um mir einen leichten Mail-, Web-, Nextcloud-, etc. Server zu installieren und weil ich einfach Bock drauf hatte. Das alles hier ist nicht in Stein gemeißelt, sondern eher ein work in progress aber soweit läuft es schonmal.

Also los gehts.

1. Eisfair als VM istallieren
2. DHCPC-Paket in der VM installieren
3. RSYNC-Paket in der VM installieren
4. SSH-Zugriff einrichten.

Jetzt müssen wir uns das Root-Dateisystem auf den Proxmox sichern.

```
rsync -aAXv --
exclude={/dev,/proc,/sys,/tmp,/run,/mnt,/media,/lost+found,/boot,/var/log/*,
/var/tmp/*,/usr/lib/firmware,/usr/lib/modules,/var/lib/alt-
kernel,/var/lib/eisman/*.db,/var/lib/eisman/info/*,System.map*}
root@<EISFAIR-IP>:/ /path/to/rootfs/
```

Jetzt haben wir das grundlegende Root-Dateisystem des eisfair und die VM wird eigentlich nicht mehr benötigt.

Damit später im LXC-Container die Funktionstasten ordnungsgemäß funktionieren einfach die folgenden zwei Befehle ausführen.

```
echo "export TERM='xterm-256color'" >> /path/to/rootfs/root/.bashrc
echo "[[ -f ~/.bashrc ]] && . ~/.bashrc" >>
/path/to/rootfs/root/.bash_profile
```

Im Root-Dateisystem noch die folgenden Gerätedaten anlegen.

- -console
- -null
- -tty

Das hier jetzt ohne Gewähr, ich habe mir die entsprechenden „Dateien“ aus einem anderen tar-Archiv kopiert. Ob das überhaupt notwendig ist und/oder ob das anders gemacht werden kann weiß ich noch nicht.

```
sudo mknod /path/to/rootfs/dev/console c 5 1
sudo mknod /path/to/rootfs/dev/null c 1 3
sudo mknod /path/to/rootfs/dev/tty c 5 0
```

Rechte setzen

```
sudo chmod 600 /path/to/rootfs/dev/console
sudo chmod 666 /path/to/rootfs/dev/null
sudo chmod 600 /path/to/rootfs/dev/tty
```

Jetzt ist das Eisfair root-fs soweit vorbereitet, dass wir es packen können

```
tar -czvf eisfair.tar.gz -C /path/to/rootfs .
```

Somit haben wir ein LXC-Template für den Proxmox erstellt. Dieses Template können wir jetzt z.B. nach **/var/lib/vz/template/cache/** kopieren.

Der Einfachheit halber gib es ein fertiges Image hier.

ACHTUNG: die Passwörter der User „eis“, „halt“ und „reboot“ habe ich entweder gelöscht oder sie heißen eis, halt und reboot

Damit wir das Template auch installieren und auch die Netzwerkeinstellungen, etc. im Proxmox konfigurieren können müssen wir auch noch ein paar Dateien auf dem Proxmox-Host anlegen/ändern.

Auf dem Proxmox Host im Verzeichnis **/usr/share/lcx/config/** die Datei **eisfair.common.conf** mit folgenden Inhalt anlegen.

/usr/share/lcx/config/eisfair.common.conf

```
# This derives from the global common config
lxc.include = /usr/share/lxc/config/common.conf


# If you wish to allow mounting block filesystems, then use the
following
# line instead, and make sure to grant access to the block device
and/or loop
# devices below in lxc.cgroup.devices.allow.
```

```
#lxc.apparmor.profile = lxc-container-default-with-mounting

# Extra cgroup device access
## rtc
lxc.cgroup.devices.allow = c 254:0 rm
## tun
lxc.cgroup.devices.allow = c 10:200 rwm
## hpet
lxc.cgroup.devices.allow = c 10:228 rwm
## kvm
lxc.cgroup.devices.allow = c 10:232 rwm
```

Und auch noch die **eisfair.userns.conf**

/usr/share/lxc/config/eisfair.userns.conf

```
# This derives from the global userns config
lxc.include = /usr/share/lxc/config/userns.conf
```

Sucht in der Datei **/usr/share/perl5/PVE/LXC/Config.pm** nach **enum ⇒ [qw(debian devuan ubuntu centos fedora opensuse archlinux alpine gentoo nixos unmanaged)],**

und ersetzt diese Zeile durch **enum ⇒ [qw(debian devuan eisfair ubuntu centos fedora opensuse archlinux alpine gentoo nixos unmanaged)],**

Die entsprechenden Zeilen sind farblich hervorgehoben, (Zeile 483)

usr/share/perl5/PVE/LXC/Config.pm

```
 1. package PVE::LXC::Config;
 2.
 3. use strict;
 4. use warnings;
 5.
 6. use Fcntl qw(O_RDONLY);
 7.
 8. use PVE::AbstractConfig;
 9. use PVE::Cluster qw(cfs_register_file);
10. use PVE::DataCenterConfig;
11. use PVE::GuestHelpers;
12. use PVE::INotify;
13. use PVE::JSONSchema qw(get_standard_option);
14. use PVE::Tools;
15.
16. use PVE::LXC;
17. use PVE::LXC::Tools;
18.
```

```
19. use base qw(PVE::AbstractConfig);
20.
21. use constant {
22.     FIFREEZE => 0xc0045877,
23.     FITHAW   => 0xc0045878,
24. };
25.
26. my $have_sdn;
27. eval {
28.     require PVE::Network::SDN::Vnets;
29.     $have_sdn = 1;
30. };
31.
32. my $nodename = PVE::INotify::nodename();
33. my $lock_handles =  {};
34. my $lockdir = "/run/lock/lxc";
35. mkdir $lockdir;
36. mkdir "/etc/pve/nodes/$nodename/lxc";
37. my $MAX_MOUNT_POINTS = 256;
38. my $MAX_UNUSED_DISKS = $MAX_MOUNT_POINTS;
39. my $MAX_DEVICES = 256;
40.
41. # BEGIN implemented abstract methods from PVE::AbstractConfig
42.
43. sub guest_type {
44.     return "CT";
45. }
46.
47. sub __config_max_unused_disks {
48.     my ($class) = @_;
49.
50.     return $MAX_UNUSED_DISKS;
51. }
52.
53. sub config_file_lock {
54.     my ($class, $vmid) = @_;
55.
56.     return "$lockdir/pve-config-${vmid}.lock";
57. }
58.
59. sub cfs_config_path {
60.     my ($class, $vmid, $node) = @_;
61.
62.     $node = $nodename if !$node;
63.     return "nodes/$node/lxc/$vmid.conf";
64. }
65.
66. sub mountpoint_backup_enabled {
67.     my ($class, $mp_key, $mountpoint) = @_;
68.
69.     my $enabled;
```

```
70.     my $reason;
71.
72.     if ($mp_key eq 'rootfs') {
73.   $enabled = 1;
74.     $reason = 'rootfs';
75.     } elsif ($mountpoint->{type} ne 'volume') {
76.     $enabled = 0;
77.     $reason = 'not a volume';
78.     } elsif ($mountpoint->{backup}) {
79.     $enabled = 1;
80.     $reason = 'enabled';
81.     } else {
82.     $enabled = 0;
83.     $reason = 'disabled';
84.     }
85.     return wantarray ? ($enabled, $reason) : $enabled;
86. }
87.
88. sub has_feature {
89.     my ($class, $feature, $conf, $storecfg, $snapname, $running,
    $backup_only) = @_;
90.     my $err;
91.
92.     my $opts;
93.     if ($feature eq 'copy' || $feature eq 'clone') {
94.     $opts = {'valid_target_formats' => ['raw', 'subvol']};
95.     }
96.
97.     $class->foreach_volume($conf, sub {
98.     my ($ms, $mountpoint) = @_;
99.
100.     return if $err; # skip further test
101.     return if $backup_only &&
    !$class->mountpoint_backup_enabled($ms, $mountpoint);
102.
103.     $err = 1 if !PVE::Storage::volume_has_feature(
104.         $storecfg, $feature, $mountpoint->{volume}, $snapname,
    $running, $opts);
105.     });
106.
107.     return $err ? 0 : 1;
108. }
109.
110. sub __snapshot_save_vmstate {
111.     my ($class, $vmid, $conf, $snapname, $storecfg) = @_;
112.     die "implement me - snapshot_save_vmstate\n";
113. }
114.
115. sub __snapshot_activate_storages {
116.     my ($class, $conf, $include_vmstate) = @_;
117.
```

```
118.     my $storecfg = PVE::Storage::config();
119.     my $opts = $include_vmstate ? { 'extra_keys' => ['vmstate'] }
     : {};
120.     my $storage_hash = {};
121.
122.     $class->foreach_volume_full($conf, $opts, sub {
123.     my ($vs, $mountpoint) = @_;
124.
125.     return if $mountpoint->{type} ne 'volume';
126.
127.     my ($storeid) =
     PVE::Storage::parse_volume_id($mountpoint->{volume});
128.     $storage_hash->{$storeid} = 1;
129.     });
130.
131.     PVE::Storage::activate_storage_list($storecfg, [ sort keys
     $storage_hash->%* ]);
132. }
133.
134. sub __snapshot_check_running {
135.     my ($class, $vmid) = @_;
136.     return PVE::LXC::check_running($vmid);
137. }
138.
139. sub __snapshot_check_freeze_needed {
140.     my ($class, $vmid, $config, $save_vmstate) = @_;
141.
142.     my $ret = $class->__snapshot_check_running($vmid);
143.     return ($ret, $ret);
144. }
145.
146. # implements similar functionality to fsfreeze(8)
147. sub fsfreeze_mountpoint {
148.     my ($path, $thaw) = @_;
149.
150.     my $op = $thaw ? 'thaw' : 'freeze';
151.     my $ioctl = $thaw ? FITHAW : FIFREEZE;
152.
153.     sysopen my $fd, $path, O_RDONLY or die "failed to open $path:
     $!\n";
154.     my $ioctl_err;
155.     if (!ioctl($fd, $ioctl, 0)) {
156.     $ioctl_err = "$!";
157.     }
158.     close($fd);
159.     die "fs$op '$path' failed - $ioctl_err\n" if defined
     $ioctl_err;
160. }
161.
162. sub __snapshot_freeze {
163.     my ($class, $vmid, $unfreeze) = @_;
```

```
164.
165.     my $conf = $class->load_config($vmid);
166.     my $storagecfg = PVE::Storage::config();
167.
168.     my $freeze_mps = [];
169.     $class->foreach_volume($conf, sub {
170.     my ($ms, $mountpoint) = @_;
171.
172.     return if $mountpoint->{type} ne 'volume';
173.
174.     if (PVE::Storage::volume_snapshot_needs_fsfreeze($storagecfg,
    $mountpoint->{volume})) {
175.         push @$freeze_mps, $mountpoint->{mp};
176.     }
177.     });
178.
179.     my $freeze_mountpoints = sub {
180.     my ($thaw) = @_;
181.
182.     return if scalar(@$freeze_mps) == 0;
183.
184.     my $pid = PVE::LXC::find_lxc_pid($vmid);
185.
186.     for my $mp (@$freeze_mps) {
187.         eval{ fsfreeze_mountpoint("/proc/${pid}/root/${mp}",
    $thaw); };
188.         warn $@ if $@;
189.     }
190.     };
191.
192.     if ($unfreeze) {
193.     eval { PVE::LXC::thaw($vmid); };
194.     warn $@ if $@;
195.     $freeze_mountpoints->(1);
196.     } else {
197.     PVE::LXC::freeze($vmid);
198.     PVE::LXC::sync_container_namespace($vmid);
199.     $freeze_mountpoints->(0);
200.     }
201. }
202.
203. sub __snapshot_create_vol_snapshot {
204.     my ($class, $vmid, $ms, $mountpoint, $snapname) = @_;
205.
206.     my $storecfg = PVE::Storage::config();
207.
208.     return if $snapname eq 'vzdump' &&
209.     !$class->mountpoint_backup_enabled($ms, $mountpoint);
210.
211.     PVE::Storage::volume_snapshot($storecfg,
    $mountpoint->{volume}, $snapname);
```

```
212. }
213.
214. sub __snapshot_delete_remove_drive {
215.     my ($class, $snap, $remove_drive) = @_;
216.
217.     if ($remove_drive eq 'vmstate') {
218.     die "implement me - saving vmstate\n";
219.     } else {
220.     my $value = $snap->{$remove_drive};
221.     my $mountpoint = $class->parse_volume($remove_drive, $value,
    1);
222.     delete $snap->{$remove_drive};
223.
224.     $class->add_unused_volume($snap, $mountpoint->{volume})
225.         if $mountpoint && ($mountpoint->{type} eq 'volume');
226.     }
227. }
228.
229. sub __snapshot_delete_vmstate_file {
230.     my ($class, $snap, $force) = @_;
231.
232.     die "implement me - saving vmstate\n";
233. }
234.
235. sub __snapshot_delete_vol_snapshot {
236.     my ($class, $vmid, $ms, $mountpoint, $snapname, $unused) = @_;
237.
238.     return if $snapname eq 'vzdump' &&
239.     !$class->mountpoint_backup_enabled($ms, $mountpoint);
240.
241.     my $storecfg = PVE::Storage::config();
242.     PVE::Storage::volume_snapshot_delete($storecfg,
    $mountpoint->{volume}, $snapname);
243.     push @$unused, $mountpoint->{volume};
244. }
245.
246. sub __snapshot_rollback_vol_possible {
247.     my ($class, $mountpoint, $snapname, $blockers) = @_;
248.
249.     my $storecfg = PVE::Storage::config();
250.     PVE::Storage::volume_rollback_is_possible(
251.     $storecfg,
252.     $mountpoint->{volume},
253.     $snapname,
254.     $blockers,
255.     );
256. }
257.
258. sub __snapshot_rollback_vol_rollback {
259.     my ($class, $mountpoint, $snapname) = @_;
260.
```

```
261.     my $storecfg = PVE::Storage::config();
262.     PVE::Storage::volume_snapshot_rollback($storecfg,
    $mountpoint->{volume}, $snapname);
263. }
264.
265. sub __snapshot_rollback_vm_stop {
266.     my ($class, $vmid) = @_;
267.
268.     PVE::LXC::vm_stop($vmid, 1)
269.     if $class->__snapshot_check_running($vmid);
270. }
271.
272. sub __snapshot_rollback_vm_start {
273.     my ($class, $vmid, $vmstate, $data);
274.
275.     die "implement me - save vmstate\n";
276. }
277.
278. sub __snapshot_rollback_get_unused {
279.     my ($class, $conf, $snap) = @_;
280.
281.     my $unused = [];
282.
283.     $class->foreach_volume($conf, sub {
284.     my ($vs, $volume) = @_;
285.
286.     return if $volume->{type} ne 'volume';
287.
288.     my $found = 0;
289.     my $volid = $volume->{volume};
290.
291.     $class->foreach_volume($snap, sub {
292.         my ($ms, $mountpoint) = @_;
293.
294.         return if $found;
295.         return if ($mountpoint->{type} ne 'volume');
296.
297.         $found = 1
298.         if ($mountpoint->{volume} && $mountpoint->{volume} eq
    $volid);
299.     });
300.
301.     push @$unused, $volid if !$found;
302.     });
303.
304.     return $unused;
305. }
306.
307. # END implemented abstract methods from PVE::AbstractConfig
308.
309. # BEGIN JSON config code
```

```
310.
311. cfs_register_file('/lxc/', \&parse_pct_config,
     \&write_pct_config);
312.
313.
314. my $valid_mount_option_re =
     qr/(discard|lazytime|noatime|nodev|noexec|nosuid)/;
315.
316. sub is_valid_mount_option {
317.     my ($option) = @_;
318.     return $option =~ $valid_mount_option_re;
319. }
320.
321. my $rootfs_desc = {
322.     volume => {
323.     type => 'string',
324.     default_key => 1,
325.     format => 'pve-lxc-mp-string',
326.     format_description => 'volume',
327.     description => 'Volume, device or directory to mount into the
     container.',
328.     },
329.     size => {
330.     type => 'string',
331.     format => 'disk-size',
332.     format_description => 'DiskSize',
333.     description => 'Volume size (read only value).',
334.     optional => 1,
335.     },
336.     acl => {
337.     type => 'boolean',
338.     description => 'Explicitly enable or disable ACL support.',
339.     optional => 1,
340.     },
341.     mountoptions => {
342.     optional => 1,
343.     type => 'string',
344.     description => 'Extra mount options for rootfs/mps.',
345.     format_description => 'opt[;opt...]',
346.     pattern =>
     qr/$valid_mount_option_re(;$valid_mount_option_re)*/,
347.     },
348.     ro => {
349.     type => 'boolean',
350.     description => 'Read-only mount point',
351.     optional => 1,
352.     },
353.     quota => {
354.     type => 'boolean',
355.     description => 'Enable user quotas inside the container (not
     supported with zfs subvolumes)',
```

```
356.     optional => 1,
357.     },
358.     replicate => {
359.     type => 'boolean',
360.     description => 'Will include this volume to a storage replica
    job.',
361.     optional => 1,
362.     default => 1,
363.     },
364.     shared => {
365.     type => 'boolean',
366.     description => 'Mark this non-volume mount point as available
    on multiple nodes (see \'nodes\')',
367.     verbose_description => "Mark this non-volume mount point as
    available on all nodes.\n\nWARNING: This option does not share the
    mount point automatically, it assumes it is shared already!",
368.     optional => 1,
369.     default => 0,
370.     },
371. };
372.
373. PVE::JSONSchema::register_standard_option('pve-ct-rootfs', {
374.     type => 'string', format => $rootfs_desc,
375.     description => "Use volume as container root.",
376.     optional => 1,
377. });
378.
379. # IP address with optional interface suffix for link local ipv6
    addresses
380. PVE::JSONSchema::register_format('lxc-ip-with-ll-iface',
    \&verify_ip_with_ll_iface);
381. sub verify_ip_with_ll_iface {
382.     my ($addr, $noerr) = @_;
383.
384.     if (my ($addr, $iface) = ($addr =~ /^(fe80:[^%]+)%(.*)$/)) {
385.     if (PVE::JSONSchema::pve_verify_ip($addr, 1)
386.         && PVE::JSONSchema::pve_verify_iface($iface, 1))
387.     {
388.         return $addr;
389.     }
390.     }
391.
392.     return PVE::JSONSchema::pve_verify_ip($addr, $noerr);
393. }
394.
395.
396. my $features_desc = {
397.     mount => {
398.     optional => 1,
399.     type => 'string',
400.     description => "Allow mounting file systems of specific
```

```
    types."
401.         ." This should be a list of file system types as used with
    the mount command."
402.         ." Note that this can have negative effects on the
    container's security."
403.         ." With access to a loop device, mounting a file can
    circumvent the mknod"
404.         ." permission of the devices cgroup, mounting an NFS file
    system can"
405.         ." block the host's I/O completely and prevent it from
    rebooting, etc.",
406.     format_description => 'fstype;fstype;...',
407.     pattern => qr/[a-zA-Z0-9_; ]+/,
408.     },
409.     nesting => {
410.     optional => 1,
411.     type => 'boolean',
412.     default => 0,
413.     description => "Allow nesting."
414.         ." Best used with unprivileged containers with additional
    id mapping."
415.         ." Note that this will expose procfs and sysfs contents of
    the host"
416.         ." to the guest.",
417.     },
418.     keyctl => {
419.     optional => 1,
420.     type => 'boolean',
421.     default => 0,
422.     description => "For unprivileged containers only: Allow the
    use of the keyctl() system call."
423.         ." This is required to use docker inside a container."
424.         ." By default unprivileged containers will see this system
    call as non-existent."
425.         ." This is mostly a workaround for systemd-networkd, as it
    will treat it as a fatal"
426.         ." error when some keyctl() operations are denied by the
    kernel due to lacking permissions."
427.         ." Essentially, you can choose between running systemd-
    networkd or docker.",
428.     },
429.     fuse => {
430.     optional => 1,
431.     type => 'boolean',
432.     default => 0,
433.     description => "Allow using 'fuse' file systems in a
    container."
434.         ." Note that interactions between fuse and the freezer
    cgroup can potentially cause I/O deadlocks.",
435.     },
436.     mknod => {
```

```
437.     optional => 1,
438.     type => 'boolean',
439.     default => 0,
440.     description => "Allow unprivileged containers to use mknod()
     to add certain device nodes."
441.         ." This requires a kernel with seccomp trap to user space
     support (5.3 or newer)."
442.         ." This is experimental.",
443.     },
444.     force_rw_sys => {
445.     optional => 1,
446.     type => 'boolean',
447.     default => 0,
448.     description => "Mount /sys in unprivileged containers as `rw`
     instead of `mixed`."
449.         ." This can break networking under newer (>= v245)
     systemd-network use."
450.     },
451. };
452.
453. my $confdesc = {
454.     lock => {
455.     optional => 1,
456.     type => 'string',
457.     description => "Lock/unlock the container.",
458.     enum => [qw(backup create destroyed disk fstrim migrate
     mounted rollback snapshot snapshot-delete)],
459.     },
460.     onboot => {
461.     optional => 1,
462.     type => 'boolean',
463.     description => "Specifies whether a container will be started
     during system bootup.",
464.     default => 0,
465.     },
466.     startup => get_standard_option('pve-startup-order'),
467.     template => {
468.     optional => 1,
469.     type => 'boolean',
470.     description => "Enable/disable Template.",
471.     default => 0,
472.     },
473.     arch => {
474.     optional => 1,
475.     type => 'string',
476.     enum => ['amd64', 'i386', 'arm64', 'armhf', 'riscv32',
     'riscv64'],
477.     description => "OS architecture type.",
478.     default => 'amd64',
479.     },
480.     ostype => {
```

```
481.     optional => 1,
482.     type => 'string',
483.     enum => [qw(debian devuan eisfair ubuntu centos fedora
    opensuse archlinux alpine gentoo nixos unmanaged)],
484.     description => "OS type. This is used to setup configuration
    inside the container, and corresponds to lxc setup scripts in
    /usr/share/lxc/config/<ostype>.common.conf. Value 'unmanaged' can
    be used to skip and OS specific setup.",
485.     },
486.     console => {
487.     optional => 1,
488.     type => 'boolean',
489.     description => "Attach a console device (/dev/console) to the
    container.",
490.     default => 1,
491.     },
492.     tty => {
493.     optional => 1,
494.     type => 'integer',
495.     description => "Specify the number of tty available to the
    container",
496.     minimum => 0,
497.     maximum => 6,
498.     default => 2,
499.     },
500.     cores => {
501.     optional => 1,
502.     type => 'integer',
503.     description => "The number of cores assigned to the container.
    A container can use all available cores by default.",
504.     minimum => 1,
505.     maximum => 8192,
506.     },
507.     cpulimit => {
508.     optional => 1,
509.     type => 'number',
510.     description => "Limit of CPU usage.\n\nNOTE: If the computer
    has 2 CPUs, it has a total of '2' CPU time. Value '0' indicates no
    CPU limit.",
511.     minimum => 0,
512.     maximum => 8192,
513.     default => 0,
514.     },
515.     cpuunits => {
516.     optional => 1,
517.     type => 'integer',
518.     description => "CPU weight for a container, will be clamped to
    [1, 10000] in cgroup v2.",
519.     verbose_description => "CPU weight for a container. Argument
    is used in the kernel fair "
520.         ."scheduler. The larger the number is, the more CPU time
```

```
    this container gets. Number "
521.         ."is relative to the weights of all the other running
    guests.",
522.     minimum => 0,
523.     maximum => 500000,
524.     default => 'cgroup v1: 1024, cgroup v2: 100',
525.     },
526.     memory => {
527.     optional => 1,
528.     type => 'integer',
529.     description => "Amount of RAM for the container in MB.",
530.     minimum => 16,
531.     default => 512,
532.     },
533.     swap => {
534.     optional => 1,
535.     type => 'integer',
536.     description => "Amount of SWAP for the container in MB.",
537.     minimum => 0,
538.     default => 512,
539.     },
540.     hostname => {
541.     optional => 1,
542.     description => "Set a host name for the container.",
543.     type => 'string', format => 'dns-name',
544.     maxLength => 255,
545.     },
546.     description => {
547.     optional => 1,
548.     type => 'string',
549.     description => "Description for the Container. Shown in the
    web-interface CT's summary."
550.         ." This is saved as comment inside the configuration
    file.",
551.     maxLength => 1024 * 8,
552.     },
553.     searchdomain => {
554.     optional => 1,
555.     type => 'string', format => 'dns-name-list',
556.     description => "Sets DNS search domains for a container.
    Create will automatically use the setting from the host if you
    neither set searchdomain nor nameserver.",
557.     },
558.     nameserver => {
559.     optional => 1,
560.     type => 'string', format => 'lxc-ip-with-ll-iface-list',
561.     description => "Sets DNS server IP address for a container.
    Create will automatically use the setting from the host if you
    neither set searchdomain nor nameserver.",
562.     },
563.     timezone => {
```

```
564.     optional => 1,
565.     type => 'string', format => 'pve-ct-timezone',
566.     description => "Time zone to use in the container. If option
     isn't set, then nothing will be done. Can be set to 'host' to
     match the host time zone, or an arbitrary time zone option from
     /usr/share/zoneinfo/zone.tab",
567.     },
568.     rootfs => get_standard_option('pve-ct-rootfs'),
569.     parent => {
570.     optional => 1,
571.     type => 'string', format => 'pve-configid',
572.     maxLength => 40,
573.     description => "Parent snapshot name. This is used internally,
     and should not be modified.",
574.     },
575.     snaptime => {
576.     optional => 1,
577.     description => "Timestamp for snapshots.",
578.     type => 'integer',
579.     minimum => 0,
580.     },
581.     cmode => {
582.     optional => 1,
583.     description => "Console mode. By default, the console command
     tries to open a connection to one of the available tty devices. By
     setting cmode to 'console' it tries to attach to /dev/console
     instead. If you set cmode to 'shell', it simply invokes a shell
     inside the container (no login).",
584.     type => 'string',
585.     enum => ['shell', 'console', 'tty'],
586.     default => 'tty',
587.     },
588.     protection => {
589.     optional => 1,
590.     type => 'boolean',
591.     description => "Sets the protection flag of the container.
     This will prevent the CT or CT's disk remove/update operation.",
592.     default => 0,
593.     },
594.     unprivileged => {
595.     optional => 1,
596.     type => 'boolean',
597.     description => "Makes the container run as unprivileged user.
     (Should not be modified manually.)",
598.     default => 0,
599.     },
600.     features => {
601.     optional => 1,
602.     type => 'string',
603.     format => $features_desc,
604.     description => "Allow containers access to advanced
```

```
    features.",
605.     },
606.     hookscript => {
607.     optional => 1,
608.     type => 'string',
609.     format => 'pve-volume-id',
610.     description => 'Script that will be executed during various
    steps in the containers lifetime.',
611.     },
612.     tags => {
613.     type => 'string', format => 'pve-tag-list',
614.     description => 'Tags of the Container. This is only meta
    information.',
615.     optional => 1,
616.     },
617.     debug => {
618.     optional => 1,
619.     type => 'boolean',
620.     description => "Try to be more verbose. For now this only
    enables debug log-level on start.",
621.     default => 0,
622.     },
623. };
624.
625. my $valid_lxc_conf_keys = {
626.     'lxc.apparmor.profile' => 1,
627.     'lxc.apparmor.allow_incomplete' => 1,
628.     'lxc.apparmor.allow_nesting' => 1,
629.     'lxc.apparmor.raw' => 1,
630.     'lxc.selinux.context' => 1,
631.     'lxc.include' => 1,
632.     'lxc.arch' => 1,
633.     'lxc.uts.name' => 1,
634.     'lxc.signal.halt' => 1,
635.     'lxc.signal.reboot' => 1,
636.     'lxc.signal.stop' => 1,
637.     'lxc.init.cmd' => 1,
638.     'lxc.pty.max' => 1,
639.     'lxc.console.logfile' => 1,
640.     'lxc.console.path' => 1,
641.     'lxc.tty.max' => 1,
642.     'lxc.devtty.dir' => 1,
643.     'lxc.hook.autodev' => 1,
644.     'lxc.autodev' => 1,
645.     'lxc.kmsg' => 1,
646.     'lxc.mount.fstab' => 1,
647.     'lxc.mount.entry' => 1,
648.     'lxc.mount.auto' => 1,
649.     'lxc.rootfs.path' => 'lxc.rootfs.path is auto generated from
    rootfs',
650.     'lxc.rootfs.mount' => 1,
```

```
    'lxc.rootfs.options' => 'lxc.rootfs.options is not supported'
.
                            ', please use mount point options in
the "rootfs" key',
    # lxc.cgroup.*
    # lxc.prlimit.*
    # lxc.net.*
    'lxc.cap.drop' => 1,
    'lxc.cap.keep' => 1,
    'lxc.seccomp.profile' => 1,
    'lxc.seccomp.notify.proxy' => 1,
    'lxc.seccomp.notify.cookie' => 1,
    'lxc.idmap' => 1,
    'lxc.hook.pre-start' => 1,
    'lxc.hook.pre-mount' => 1,
    'lxc.hook.mount' => 1,
    'lxc.hook.start' => 1,
    'lxc.hook.stop' => 1,
    'lxc.hook.post-stop' => 1,
    'lxc.hook.clone' => 1,
    'lxc.hook.destroy' => 1,
    'lxc.hook.version' => 1,
    'lxc.log.level' => 1,
    'lxc.log.file' => 1,
    'lxc.start.auto' => 1,
    'lxc.start.delay' => 1,
    'lxc.start.order' => 1,
    'lxc.group' => 1,
    'lxc.environment' => 1,

    # All these are namespaced via CLONE_NEWIPC (see
namespaces(7)).
    'lxc.sysctl.fs.mqueue' => 1,
    'lxc.sysctl.kernel.msgmax' => 1,
    'lxc.sysctl.kernel.msgmnb' => 1,
    'lxc.sysctl.kernel.msgmni' => 1,
    'lxc.sysctl.kernel.sem' => 1,
    'lxc.sysctl.kernel.shmall' => 1,
    'lxc.sysctl.kernel.shmmax' => 1,
    'lxc.sysctl.kernel.shmmni' => 1,
    'lxc.sysctl.kernel.shm_rmid_forced' => 1,
};

my $deprecated_lxc_conf_keys = {
    # Deprecated (removed with lxc 3.0):
    'lxc.aa_profile'           => 'lxc.apparmor.profile',
    'lxc.aa_allow_incomplete'  => 'lxc.apparmor.allow_incomplete',
    'lxc.console'              => 'lxc.console.path',
    'lxc.devttydir'            => 'lxc.tty.dir',
    'lxc.haltsignal'           => 'lxc.signal.halt',
    'lxc.rebootsignal'         => 'lxc.signal.reboot',
```

```
699.     'lxc.stopsignal'           => 'lxc.signal.stop',
700.     'lxc.id_map'               => 'lxc.idmap',
701.     'lxc.init_cmd'             => 'lxc.init.cmd',
702.     'lxc.loglevel'             => 'lxc.log.level',
703.     'lxc.logfile'              => 'lxc.log.file',
704.     'lxc.mount'                => 'lxc.mount.fstab',
705.     'lxc.network.type'         => 'lxc.net.INDEX.type',
706.     'lxc.network.flags'        => 'lxc.net.INDEX.flags',
707.     'lxc.network.link'         => 'lxc.net.INDEX.link',
708.     'lxc.network.mtu'          => 'lxc.net.INDEX.mtu',
709.     'lxc.network.name'         => 'lxc.net.INDEX.name',
710.     'lxc.network.hwaddr'       => 'lxc.net.INDEX.hwaddr',
711.     'lxc.network.ipv4'         => 'lxc.net.INDEX.ipv4.address',
712.     'lxc.network.ipv4.gateway' => 'lxc.net.INDEX.ipv4.gateway',
713.     'lxc.network.ipv6'         => 'lxc.net.INDEX.ipv6.address',
714.     'lxc.network.ipv6.gateway' => 'lxc.net.INDEX.ipv6.gateway',
715.     'lxc.network.script.up'    => 'lxc.net.INDEX.script.up',
716.     'lxc.network.script.down'  => 'lxc.net.INDEX.script.down',
717.     'lxc.pts'                  => 'lxc.pty.max',
718.     'lxc.se_context'           => 'lxc.selinux.context',
719.     'lxc.seccomp'              => 'lxc.seccomp.profile',
720.     'lxc.tty'                  => 'lxc.tty.max',
721.     'lxc.utsname'              => 'lxc.uts.name',
722. };
723.
724. sub is_valid_lxc_conf_key {
725.     my ($vmid, $key) = @_;
726.     if ($key =~ /^lxc\.limit\./) {
727.     warn "vm $vmid - $key: lxc.limit.* was renamed to
    lxc.prlimit.*\n";
728.     return 1;
729.     }
730.     if (defined(my $new_name = $deprecated_lxc_conf_keys->{$key}))
    {
731.     warn "vm $vmid - $key is deprecated and was renamed to
    $new_name\n";
732.     return 1;
733.     }
734.     my $validity = $valid_lxc_conf_keys->{$key};
735.     return $validity if defined($validity);
736.     return 1 if $key =~ /^lxc\.cgroup2?\./  # allow all cgroup
    values
737.             || $key =~ /^lxc\.prlimit\./ # allow all prlimits
738.             || $key =~ /^lxc\.net\./;    # allow custom network
    definitions
739.     return 0;
740. }
741.
742. our $netconf_desc = {
743.     type => {
744.     type => 'string',
```

```
745.     optional => 1,
746.     description => "Network interface type.",
747.     enum => [qw(veth)],
748.     },
749.     name => {
750.     type => 'string',
751.         format_description => 'string',
752.      description => 'Name of the network device as seen from
    inside the container. (lxc.network.name)',
753.     pattern => '[-_.\w\d]+',
754.     },
755.     bridge => {
756.     type => 'string',
757.     format_description => 'bridge',
758.     description => 'Bridge to attach the network device to.',
759.     pattern => '[-_.\w\d]+',
760.     optional => 1,
761.     },
762.     hwaddr => get_standard_option('mac-addr', {
763.         description => 'The interface MAC address. This is
    dynamically allocated by default, but you can set that statically
    if needed, for example to always have the same link-local IPv6
    address. (lxc.network.hwaddr)',
764.     }),
765.     mtu => {
766.     type => 'integer',
767.     description => 'Maximum transfer unit of the interface.
    (lxc.network.mtu)',
768.     minimum => 64, # minimum ethernet frame is 64 bytes
769.     maximum => 65535,
770.     optional => 1,
771.     },
772.     ip => {
773.     type => 'string',
774.     format => 'pve-ipv4-config',
775.     format_description => '(IPv4/CIDR|dhcp|manual)',
776.     description => 'IPv4 address in CIDR format.',
777.     optional => 1,
778.     },
779.     gw => {
780.     type => 'string',
781.     format => 'ipv4',
782.     format_description => 'GatewayIPv4',
783.     description => 'Default gateway for IPv4 traffic.',
784.     optional => 1,
785.     },
786.     ip6 => {
787.     type => 'string',
788.     format => 'pve-ipv6-config',
789.     format_description => '(IPv6/CIDR|auto|dhcp|manual)',
790.     description => 'IPv6 address in CIDR format.',
```

```
791.     optional => 1,
792.     },
793.     gw6 => {
794.     type => 'string',
795.     format => 'ipv6',
796.     format_description => 'GatewayIPv6',
797.     description => 'Default gateway for IPv6 traffic.',
798.     optional => 1,
799.     },
800.     firewall => {
801.     type => 'boolean',
802.     description => "Controls whether this interface's firewall
    rules should be used.",
803.     optional => 1,
804.     },
805.     tag => {
806.     type => 'integer',
807.     minimum => 1,
808.     maximum => 4094,
809.     description => "VLAN tag for this interface.",
810.     optional => 1,
811.     },
812.     trunks => {
813.     type => 'string',
814.     pattern => qr/\d+(?:;\d+)*/,
815.     format_description => 'vlanid[;vlanid...]',
816.     description => "VLAN ids to pass through the interface",
817.     optional => 1,
818.     },
819.     rate => {
820.     type => 'number',
821.     format_description => 'mbps',
822.     description => "Apply rate limiting to the interface",
823.     optional => 1,
824.     },
825.     # TODO: Rename this option and the qemu-server one to `link-
    down` for PVE 8.0
826.     link_down => {
827.     type => 'boolean',
828.     description => 'Whether this interface should be disconnected
    (like pulling the plug).',
829.     optional => 1,
830.     },
831. };
832. PVE::JSONSchema::register_format('pve-lxc-network',
    $netconf_desc);
833.
834. my $MAX_LXC_NETWORKS = 32;
835. for (my $i = 0; $i < $MAX_LXC_NETWORKS; $i++) {
836.     $confdesc->{"net$i"} = {
837.     optional => 1,
```

```
838.     type => 'string', format => $netconf_desc,
839.     description => "Specifies network interfaces for the
    container.",
840.     };
841. }
842.
843. PVE::JSONSchema::register_format('pve-ct-timezone',
    \&verify_ct_timezone);
844. sub verify_ct_timezone {
845.     my ($timezone, $noerr) = @_;
846.
847.     return if $timezone eq 'host'; # using host settings
848.
849.     PVE::JSONSchema::pve_verify_timezone($timezone);
850. }
851.
852. PVE::JSONSchema::register_format('pve-lxc-mp-string',
    \&verify_lxc_mp_string);
853. sub verify_lxc_mp_string {
854.     my ($mp, $noerr) = @_;
855.
856.     # do not allow:
857.     # /./ or /../
858.     # /. or /.. at the end
859.     # ../ at the beginning
860.
861.     if($mp =~ m@/\.\.?/@ ||
862.        $mp =~ m@/\.\.?$@ ||
863.        $mp =~ m@^\.\./@) {
864.     return undef if $noerr;
865.     die "$mp contains illegal character sequences\n";
866.     }
867.     return $mp;
868. }
869.
870. my $mp_desc = {
871.     %$rootfs_desc,
872.     backup => {
873.     type => 'boolean',
874.     description => 'Whether to include the mount point in
    backups.',
875.     verbose_description => 'Whether to include the mount point in
    backups '.
876.                     '(only used for volume mount points).',
877.     optional => 1,
878.     },
879.     mp => {
880.     type => 'string',
881.     format => 'pve-lxc-mp-string',
882.     format_description => 'Path',
883.     description => 'Path to the mount point as seen from inside
```

```
    the container '.
884.                 '(must not contain symlinks).',
885.     verbose_description => "Path to the mount point as seen from
    inside the container.\n\n".
886.                    "NOTE: Must not contain any symlinks for
    security reasons."
887.     },
888. };
889. PVE::JSONSchema::register_format('pve-ct-mountpoint', $mp_desc);
890.
891. my $unused_desc = {
892.     volume => {
893.     type => 'string',
894.     default_key => 1,
895.     format => 'pve-volume-id',
896.     format_description => 'volume',
897.     description => 'The volume that is not used currently.',
898.     }
899. };
900.
901. for (my $i = 0; $i < $MAX_MOUNT_POINTS; $i++) {
902.     $confdesc->{"mp$i"} = {
903.     optional => 1,
904.     type => 'string', format => $mp_desc,
905.     description => "Use volume as container mount point. Use the
    special " .
906.         "syntax STORAGE_ID:SIZE_IN_GiB to allocate a new volume.",
907.     };
908. }
909.
910. for (my $i = 0; $i < $MAX_UNUSED_DISKS; $i++) {
911.     $confdesc->{"unused$i"} = {
912.     optional => 1,
913.     type => 'string', format => $unused_desc,
914.     description => "Reference to unused volumes. This is used
    internally, and should not be modified manually.",
915.     }
916. }
917.
918. PVE::JSONSchema::register_format('pve-lxc-dev-string',
    \&verify_lxc_dev_string);
919. sub verify_lxc_dev_string {
920.     my ($dev, $noerr) = @_;
921.
922.     # do not allow /./ or /../ or /.$ or /..$
923.     # enforce /dev/ at the beginning
924.
925.     if (
926.     $dev =~ m@/\.\.?(?:/|$)@ ||
927.     $dev !~ m!^/dev/!
928.     ) {
```

```
929.     return undef if $noerr;
930.     die "$dev is not a valid device path\n";
931.     }
932.
933.     return $dev;
934. }
935.
936. my $dev_desc = {
937.     path => {
938.     optional => 1,
939.     type => 'string',
940.     default_key => 1,
941.     format => 'pve-lxc-dev-string',
942.     format_description => 'Path',
943.     description => 'Device to pass through to the container',
944.     verbose_description => 'Path to the device to pass through to
    the container',
945.     },
946.     mode => {
947.     optional => 1,
948.     type => 'string',
949.     pattern => '0[0-7]{3}',
950.     format_description => 'Octal access mode',
951.     description => 'Access mode to be set on the device node',
952.     },
953.     uid => {
954.     optional => 1,
955.     type => 'integer',
956.     minimum => 0,
957.     description => 'User ID to be assigned to the device node',
958.     },
959.     gid => {
960.     optional => 1,
961.     type => 'integer',
962.     minimum => 0,
963.     description => 'Group ID to be assigned to the device node',
964.     },
965.     'deny-write' => {
966.     optional => 1,
967.     type => 'boolean',
968.     description => 'Deny the container to write to the device',
969.     default => 0,
970.     },
971. };
972.
973. for (my $i = 0; $i < $MAX_DEVICES; $i++) {
974.     $confdesc->{"dev$i"} = {
975.     optional => 1,
976.     type => 'string', format => $dev_desc,
977.     description => "Device to pass through to the container",
978.     }
```

```
979. }
980.
981. sub parse_pct_config {
982.     my ($filename, $raw, $strict) = @_;
983.
984.     return undef if !defined($raw);
985.
986.     my $res = {
987.     digest => Digest::SHA::sha1_hex($raw),
988.     snapshots => {},
989.     pending => {},
990.     };
991.
992.     my $handle_error = sub {
993.     my ($msg) = @_;
994.
995.     if ($strict) {
996.         die $msg;
997.     } else {
998.         warn $msg;
999.     }
1000.     };
1001.
1002.     $filename =~ m|/lxc/(\d+).conf$|
1003.     || die "got strange filename '$filename'";
1004.
1005.     my $vmid = $1;
1006.
1007.     my $conf = $res;
1008.     my $descr = '';
1009.     my $section = '';
1010.
1011.     my @lines = split(/\n/, $raw);
1012.     foreach my $line (@lines) {
1013.     next if $line =~ m/^\s*$/;
1014.
1015.     if ($line =~ m/^\[pve:pending\]\s*$/i) {
1016.         $section = 'pending';
1017.         $conf->{description} = $descr if $descr;
1018.         $descr = '';
1019.         $conf = $res->{$section} = {};
1020.         next;
1021.     } elsif ($line =~ m/^\[([a-z][a-z0-9_\-]+)\]\s*$/i) {
1022.         $section = $1;
1023.         $conf->{description} = $descr if $descr;
1024.         $descr = '';
1025.         $conf = $res->{snapshots}->{$section} = {};
1026.         next;
1027.     }
1028.
1029.     if ($line =~ m/^\#(.*)$/) {
```

```
1030.         $descr .= PVE::Tools::decode_text($1) . "\n";
1031.         next;
1032.     }
1033.
1034.     if ($line =~ m/^(lxc\.[a-z0-9_\-\.]+)(:|\s*=)\s*(.*?)\s*$/) {
1035.         my $key = $1;
1036.         my $value = $3;
1037.         my $validity = is_valid_lxc_conf_key($vmid, $key);
1038.         if ($validity eq 1) {
1039.         push @{$conf->{lxc}}, [$key, $value];
1040.         } elsif (my $errmsg = $validity) {
1041.         $handle_error->("vm $vmid - $key: $errmsg\n");
1042.         } else {
1043.         $handle_error->("vm $vmid - unable to parse config:
      $line\n");
1044.         }
1045.     } elsif ($line =~ m/^(description):\s*(.*\S)\s*$/) {
1046.         $descr .= PVE::Tools::decode_text($2);
1047.     } elsif ($line =~ m/snapstate:\s*(prepare|delete)\s*$/) {
1048.         $conf->{snapstate} = $1;
1049.     } elsif ($line =~ m/^delete:\s*(.*\S)\s*$/) {
1050.         my $value = $1;
1051.         if ($section eq 'pending') {
1052.         $conf->{delete} = $value;
1053.         } else {
1054.         $handle_error->("vm $vmid - property 'delete' is only
      allowed in [pve:pending]\n");
1055.         }
1056.     } elsif ($line =~ m/^([a-z][a-z_]*\d*):\s*(.+?)\s*$/) {
1057.         my $key = $1;
1058.         my $value = $2;
1059.         eval { $value = PVE::LXC::Config->check_type($key,
      $value); };
1060.         $handle_error->("vm $vmid - unable to parse value of
      '$key' - $@") if $@;
1061.         $conf->{$key} = $value;
1062.     } else {
1063.         $handle_error->("vm $vmid - unable to parse config:
      $line\n");
1064.     }
1065.     }
1066.
1067.     $conf->{description} = $descr if $descr;
1068.
1069.     delete $res->{snapstate}; # just to be sure
1070.
1071.     return $res;
1072. }
1073.
1074. sub write_pct_config {
1075.     my ($filename, $conf) = @_;
```

```
1076.
1077.     delete $conf->{snapstate}; # just to be sure
1078.
1079.     my $volidlist = PVE::LXC::Config->get_vm_volumes($conf);
1080.     my $used_volids = {};
1081.     foreach my $vid (@$volidlist) {
1082.         $used_volids->{$vid} = 1;
1083.     }
1084.
1085.     # remove 'unusedX' settings if the volume is still used
1086.     foreach my $key (keys %$conf) {
1087.         my $value = $conf->{$key};
1088.         if ($key =~ m/^unused/ && $used_volids->{$value}) {
1089.             delete $conf->{$key};
1090.         }
1091.     }
1092.
1093.     my $generate_raw_config = sub {
1094.     my ($conf) = @_;
1095.
1096.     my $raw = '';
1097.
1098.     # add description as comment to top of file
1099.     my $descr = $conf->{description} || '';
1100.     foreach my $cl (split(/\n/, $descr)) {
1101.         $raw .= '#' . PVE::Tools::encode_text($cl) . "\n";
1102.     }
1103.
1104.     foreach my $key (sort keys %$conf) {
1105.         next if $key eq 'digest' || $key eq 'description' ||
1106.             $key eq 'pending' || $key eq 'snapshots' ||
1107.             $key eq 'snapname' || $key eq 'lxc';
1108.         my $value = $conf->{$key};
1109.         die "detected invalid newline inside property '$key'\n"
1110.         if $value =~ m/\n/;
1111.         $raw .= "$key: $value\n";
1112.     }
1113.
1114.     if (my $lxcconf = $conf->{lxc}) {
1115.         foreach my $entry (@$lxcconf) {
1116.         my ($k, $v) = @$entry;
1117.         $raw .= "$k: $v\n";
1118.         }
1119.     }
1120.
1121.     return $raw;
1122.     };
1123.
1124.     my $raw = &$generate_raw_config($conf);
1125.
1126.     if (scalar(keys %{$conf->{pending}})){
```

```
1127.     $raw .= "\n[pve:pending]\n";
1128.     $raw .= &$generate_raw_config($conf->{pending});
1129.     }
1130.
1131.     foreach my $snapname (sort keys %{$conf->{snapshots}}) {
1132.     $raw .= "\n[$snapname]\n";
1133.     $raw .=
      &$generate_raw_config($conf->{snapshots}->{$snapname});
1134.     }
1135.
1136.     return $raw;
1137. }
1138.
1139. sub update_pct_config {
1140.     my ($class, $vmid, $conf, $running, $param, $delete, $revert)
      = @_;
1141.
1142.     my $storage_cfg = PVE::Storage::config();
1143.
1144.     foreach my $opt (@$revert) {
1145.     delete $conf->{pending}->{$opt};
1146.     $class->remove_from_pending_delete($conf, $opt); # also remove
      from deletion queue
1147.     }
1148.
1149.     # write updates to pending section
1150.     my $modified = {}; # record modified options
1151.
1152.     foreach my $opt (@$delete) {
1153.     if (!defined($conf->{$opt}) &&
      !defined($conf->{pending}->{$opt})) {
1154.         warn "cannot delete '$opt' - not set in current
      configuration!\n";
1155.         next;
1156.     }
1157.     $modified->{$opt} = 1;
1158.     if ($opt eq 'memory' || $opt eq 'rootfs' || $opt eq 'ostype')
      {
1159.         die "unable to delete required option '$opt'\n";
1160.     } elsif ($opt =~ m/^unused(\d+)$/) {
1161.         $class->check_protection($conf, "can't remove CT $vmid
      drive '$opt'");
1162.     } elsif ($opt =~ m/^mp(\d+)$/) {
1163.         $class->check_protection($conf, "can't remove CT $vmid
      drive '$opt'");
1164.     } elsif ($opt eq 'unprivileged') {
1165.         die "unable to delete read-only option: '$opt'\n";
1166.     }
1167.     $class->add_to_pending_delete($conf, $opt);
1168.     }
1169.
```

```
1170.     my $check_content_type = sub {
1171.     my ($mp) = @_;
1172.     my $sid = PVE::Storage::parse_volume_id($mp->{volume});
1173.     my $storage_config =
      PVE::Storage::storage_config($storage_cfg, $sid);
1174.     die "storage '$sid' does not allow content type 'rootdir'
      (Container)\n"
1175.         if !$storage_config->{content}->{rootdir};
1176.     };
1177.
1178.     foreach my $opt (sort keys %$param) { # add/change
1179.     $modified->{$opt} = 1;
1180.     my $value = $param->{$opt};
1181.     if ($opt =~ m/^mp(\d+)$/ || $opt eq 'rootfs') {
1182.         $class->check_protection($conf, "can't update CT $vmid
      drive '$opt'");
1183.         my $mp = $class->parse_volume($opt, $value);
1184.         $check_content_type->($mp) if ($mp->{type} eq 'volume');
1185.     } elsif ($opt eq 'hookscript') {
1186.         PVE::GuestHelpers::check_hookscript($value);
1187.     } elsif ($opt eq 'nameserver') {
1188.         $value = PVE::LXC::verify_nameserver_list($value);
1189.     } elsif ($opt eq 'searchdomain') {
1190.         $value = PVE::LXC::verify_searchdomain_list($value);
1191.     } elsif ($opt eq 'unprivileged') {
1192.         die "unable to modify read-only option: '$opt'\n";
1193.     } elsif ($opt eq 'tags') {
1194.         $value = PVE::GuestHelpers::get_unique_tags($value);
1195.     } elsif ($opt =~ m/^net(\d+)$/) {
1196.         my $res =
      PVE::JSONSchema::parse_property_string($netconf_desc, $value);
1197.
1198.         if (my $mtu = $res->{mtu}) {
1199.         my $bridge_mtu =
      PVE::Network::read_bridge_mtu($res->{bridge});
1200.         die "$opt: MTU size '$mtu' is bigger than bridge MTU
      '$bridge_mtu'\n"
1201.             if ($mtu > $bridge_mtu);
1202.         }
1203.     } elsif ($opt =~ m/^dev(\d+)$/) {
1204.         my $device = $class->parse_device($value);
1205.
1206.         die "Path is not defined for passthrough device $opt"
1207.         if !defined($device->{path});
1208.
1209.             # Validate device
1210.
      PVE::LXC::Tools::get_device_mode_and_rdev($device->{path});
1211.     }
1212.     $conf->{pending}->{$opt} = $value;
1213.     $class->remove_from_pending_delete($conf, $opt);
```

```
1214.     }
1215.
1216.     my $changes = $class->cleanup_pending($conf);
1217.
1218.     my $errors = {};
1219.     if ($running) {
1220.     $class->vmconfig_hotplug_pending($vmid, $conf, $storage_cfg,
     $modified, $errors);
1221.     } else {
1222.     $class->vmconfig_apply_pending($vmid, $conf, $storage_cfg,
     $modified, $errors);
1223.     }
1224.
1225.     return $errors;
1226. }
1227.
1228. sub check_type {
1229.     my ($class, $key, $value) = @_;
1230.
1231.     die "unknown setting '$key'\n" if !$confdesc->{$key};
1232.
1233.     my $type = $confdesc->{$key}->{type};
1234.
1235.     if (!defined($value)) {
1236.     die "got undefined value\n";
1237.     }
1238.
1239.     if ($value =~ m/[\n\r]/) {
1240.     die "property contains a line feed\n";
1241.     }
1242.
1243.     if ($type eq 'boolean') {
1244.     return 1 if ($value eq '1') || ($value =~
     m/^(on|yes|true)$/i);
1245.     return 0 if ($value eq '0') || ($value =~
     m/^(off|no|false)$/i);
1246.     die "type check ('boolean') failed - got '$value'\n";
1247.     } elsif ($type eq 'integer') {
1248.     return int($1) if $value =~ m/^(\d+)$/;
1249.     die "type check ('integer') failed - got '$value'\n";
1250.     } elsif ($type eq 'number') {
1251.     return $value if $value =~ m/^(\d+)(\.\d+)?$/;
1252.     die "type check ('number') failed - got '$value'\n";
1253.     } elsif ($type eq 'string') {
1254.     if (my $fmt = $confdesc->{$key}->{format}) {
1255.         PVE::JSONSchema::check_format($fmt, $value);
1256.         return $value;
1257.     }
1258.     return $value;
1259.     } else {
1260.     die "internal error"
```

```
    }
}


# add JSON properties for create and set function
sub json_config_properties {
    my ($class, $prop) = @_;

    foreach my $opt (keys %$confdesc) {
    next if $opt eq 'parent' || $opt eq 'snaptime';
    next if $prop->{$opt};
    $prop->{$opt} = $confdesc->{$opt};
    }

    return $prop;
}

my $parse_ct_mountpoint_full = sub {
    my ($class, $desc, $data, $noerr) = @_;

    $data //= '';

    my $res;
    eval { $res = PVE::JSONSchema::parse_property_string($desc, $data) };
    if ($@) {
    return undef if $noerr;
    die $@;
    }

    if (defined(my $size = $res->{size})) {
    $size = PVE::JSONSchema::parse_size($size);
    if (!defined($size)) {
        return undef if $noerr;
        die "invalid size: $size\n";
    }
    $res->{size} = $size;
    }

    $res->{type} = $class->classify_mountpoint($res->{volume});

    return $res;
};

sub print_ct_mountpoint {
    my ($class, $info, $nomp) = @_;
    my $skip = [ 'type' ];
    push @$skip, 'mp' if $nomp;
    return PVE::JSONSchema::print_property_string($info, $mp_desc, $skip);
}
```

```
1310.
1311. sub print_ct_unused {
1312.     my ($class, $info) = @_;
1313.
1314.     my $skip = [ 'type' ];
1315.     return PVE::JSONSchema::print_property_string($info,
      $unused_desc, $skip);
1316. }
1317.
1318. sub parse_volume {
1319.     my ($class, $key, $volume_string, $noerr) = @_;
1320.
1321.     if ($key eq 'rootfs') {
1322.     my $res =  $parse_ct_mountpoint_full->($class, $rootfs_desc,
      $volume_string, $noerr);
1323.     $res->{mp} = '/' if defined($res);
1324.     return $res;
1325.     } elsif ($key =~ m/^mp\d+$/) {
1326.     return $parse_ct_mountpoint_full->($class, $mp_desc,
      $volume_string, $noerr);
1327.     } elsif ($key =~ m/^unused\d+$/) {
1328.     return $parse_ct_mountpoint_full->($class, $unused_desc,
      $volume_string, $noerr);
1329.     }
1330.
1331.     die "parse_volume - unknown type: $key\n" if !$noerr;
1332.
1333.     return;
1334. }
1335.
1336. sub parse_device {
1337.     my ($class, $device_string, $noerr) = @_;
1338.
1339.     my $res = eval {
      PVE::JSONSchema::parse_property_string($dev_desc, $device_string)
      };
1340.     if ($@) {
1341.     return undef if $noerr;
1342.     die $@;
1343.     }
1344.
1345.     if (!defined($res->{path})) {
1346.     return undef if $noerr;
1347.     die "Path has to be defined\n";
1348.     }
1349.
1350.     return $res;
1351. }
1352.
1353. sub print_volume {
1354.     my ($class, $key, $volume) = @_;
```

```
1355.
1356.     return $class->print_ct_unused($volume) if $key =~
      m/^unused(\d+)$/;
1357.
1358.     return $class->print_ct_mountpoint($volume, $key eq 'rootfs');
1359. }
1360.
1361. sub volid_key {
1362.     my ($class) = @_;
1363.
1364.     return 'volume';
1365. }
1366.
1367. sub print_lxc_network {
1368.     my ($class, $net) = @_;
1369.     return PVE::JSONSchema::print_property_string($net,
      $netconf_desc);
1370. }
1371.
1372. sub parse_lxc_network {
1373.     my ($class, $data) = @_;
1374.
1375.     return {} if !$data;
1376.
1377.     my $res =
      PVE::JSONSchema::parse_property_string($netconf_desc, $data);
1378.
1379.     $res->{type} = 'veth';
1380.     if (!$res->{hwaddr}) {
1381.     my $dc = PVE::Cluster::cfs_read_file('datacenter.cfg');
1382.     $res->{hwaddr} =
      PVE::Tools::random_ether_addr($dc->{mac_prefix});
1383.     }
1384.
1385.     return $res;
1386. }
1387.
1388. sub parse_features {
1389.     my ($class, $data) = @_;
1390.     return {} if !$data;
1391.     return PVE::JSONSchema::parse_property_string($features_desc,
      $data);
1392. }
1393.
1394. sub option_exists {
1395.     my ($class, $name) = @_;
1396.
1397.     return defined($confdesc->{$name});
1398. }
1399. # END JSON config code
1400.
```

```
1401. # takes a max memory value as KiB and returns an tuple with max
      and high values
1402. sub calculate_memory_constraints {
1403.     my ($memory) = @_;
1404.
1405.     return if !defined($memory);
1406.
1407.     # cgroup memory usage is limited by the hard 'max' limit (OOM-
      killer enforced) and the soft
1408.     # 'high' limit (cgroup processes get throttled and put under
      heavy reclaim pressure).
1409.     my $memory_max = int($memory * 1024 * 1024);
1410.     # Set the high to 1016/1024 (~99.2%) of the 'max' hard limit
      clamped to 128 MiB max, to scale
1411.     # it for the lower range while having a decent 2^x based rest
      for 2^y memory configs.
1412.     my $memory_high = $memory >= 16 * 1024 ? int(($memory - 128) *
      1024 * 1024) : int($memory * 1024 * 1016);
1413.
1414.     return ($memory_max, $memory_high);
1415. }
1416.
1417. my $LXC_FASTPLUG_OPTIONS= {
1418.     'description' => 1,
1419.     'onboot' => 1,
1420.     'startup' => 1,
1421.     'protection' => 1,
1422.     'hostname' => 1,
1423.     'hookscript' => 1,
1424.     'cores' => 1,
1425.     'tags' => 1,
1426.     'lock' => 1,
1427. };
1428.
1429. sub vmconfig_hotplug_pending {
1430.     my ($class, $vmid, $conf, $storecfg, $selection, $errors) =
      @_;
1431.
1432.     my $pid = PVE::LXC::find_lxc_pid($vmid);
1433.     my $rootdir = "/proc/$pid/root";
1434.
1435.     my $add_hotplug_error = sub {
1436.     my ($opt, $msg) = @_;
1437.     $errors->{$opt} = "unable to hotplug $opt: $msg";
1438.     };
1439.
1440.     foreach my $opt (sort keys %{$conf->{pending}}) { # add/change
1441.     next if $selection && !$selection->{$opt};
1442.     if ($LXC_FASTPLUG_OPTIONS->{$opt}) {
1443.         $conf->{$opt} = delete $conf->{pending}->{$opt};
1444.     }
```

```
1445.     }
1446.
1447.     my $cgroup = PVE::LXC::CGroup->new($vmid);
1448.
1449.     # There's no separate swap size to configure, there's memory
      and "total"
1450.     # memory (iow. memory+swap). This means we have to change them
      together.
1451.     my $hotplug_memory_done;
1452.     my $hotplug_memory = sub {
1453.     my ($new_memory, $new_swap) = @_;
1454.
1455.     ($new_memory, my $new_memory_high) =
      calculate_memory_constraints($new_memory);
1456.     $new_swap = int($new_swap * 1024 * 1024) if
      defined($new_swap);
1457.     $cgroup->change_memory_limit($new_memory, $new_swap,
      $new_memory_high);
1458.
1459.     $hotplug_memory_done = 1;
1460.     };
1461.
1462.     my $pending_delete_hash =
      $class->parse_pending_delete($conf->{pending}->{delete});
1463.     # FIXME: $force deletion is not implemented for CTs
1464.     foreach my $opt (sort keys %$pending_delete_hash) {
1465.     next if $selection && !$selection->{$opt};
1466.     eval {
1467.         if ($LXC_FASTPLUG_OPTIONS->{$opt}) {
1468.         # pass
1469.         } elsif ($opt =~ m/^unused(\d+)$/) {
1470.         PVE::LXC::delete_mountpoint_volume($storecfg, $vmid,
      $conf->{$opt})
1471.             if !$class->is_volume_in_use($conf, $conf->{$opt}, 1,
      1);
1472.         } elsif ($opt eq 'swap') {
1473.         $hotplug_memory->(undef, 0);
1474.         } elsif ($opt eq 'cpulimit') {
1475.         $cgroup->change_cpu_quota(undef, undef); # reset, cgroup
      module can better decide values
1476.         } elsif ($opt eq 'cpuunits') {
1477.         $cgroup->change_cpu_shares(undef);
1478.         } elsif ($opt =~ m/^net(\d)$/) {
1479.         my $netid = $1;
1480.         PVE::Network::veth_delete("veth${vmid}i$netid");
1481.         if ($have_sdn) {
1482.             my $net =
      PVE::LXC::Config->parse_lxc_network($conf->{$opt});
1483.             print "delete ips from $opt\n";
1484.             eval {
      PVE::Network::SDN::Vnets::del_ips_from_mac($net->{bridge},
```

```
    $net->{hwaddr}, $conf->{hostname}) };
1485.             warn $@ if $@;
1486.         }
1487.         } else {
1488.         die "skip\n"; # skip non-hotpluggable opts
1489.         }
1490.     };
1491.     if (my $err = $@) {
1492.         $add_hotplug_error->($opt, $err) if $err ne "skip\n";
1493.     } else {
1494.         delete $conf->{$opt};
1495.         $class->remove_from_pending_delete($conf, $opt);
1496.     }
1497.     }
1498.
1499.     foreach my $opt (sort keys %{$conf->{pending}}) {
1500.     next if $opt eq 'delete'; # just to be sure
1501.     next if $selection && !$selection->{$opt};
1502.     my $value = $conf->{pending}->{$opt};
1503.     eval {
1504.         if ($opt eq 'cpulimit') {
1505.         my $quota = 100000 * $value;
1506.         $cgroup->change_cpu_quota(int(100000 * $value), 100000);
1507.         } elsif ($opt eq 'cpuunits') {
1508.         $cgroup->change_cpu_shares($value);
1509.         } elsif ($opt =~ m/^net(\d+)$/) {
1510.         my $netid = $1;
1511.         my $net = $class->parse_lxc_network($value);
1512.         $value = $class->print_lxc_network($net);
1513.         PVE::LXC::update_net($vmid, $conf, $opt, $net, $netid,
    $rootdir);
1514.         } elsif ($opt eq 'memory' || $opt eq 'swap') {
1515.         if (!$hotplug_memory_done) { # don't call twice if both
    opts are passed
1516.             $hotplug_memory->($conf->{pending}->{memory},
    $conf->{pending}->{swap});
1517.         }
1518.         } elsif ($opt =~ m/^mp(\d+)$/) {
1519.         if (exists($conf->{$opt})) {
1520.             die "skip\n"; # don't try to hotplug over existing mp
1521.         }
1522.
1523.         $class->apply_pending_mountpoint($vmid, $conf, $opt,
    $storecfg, 1);
1524.         # apply_pending_mountpoint modifies the value if it
    creates a new disk
1525.         $value = $conf->{pending}->{$opt};
1526.         } else {
1527.         die "skip\n"; # skip non-hotpluggable
1528.         }
1529.     };
```

```
1530.     if (my $err = $@) {
1531.         $add_hotplug_error->($opt, $err) if $err ne "skip\n";
1532.     } else {
1533.         $conf->{$opt} = $value;
1534.         delete $conf->{pending}->{$opt};
1535.     }
1536.     }
1537. }
1538.
1539. sub vmconfig_apply_pending {
1540.     my ($class, $vmid, $conf, $storecfg, $selection, $errors) =
      @_;
1541.
1542.     my $add_apply_error = sub {
1543.     my ($opt, $msg) = @_;
1544.     my $err_msg = "unable to apply pending change $opt : $msg";
1545.     $errors->{$opt} = $err_msg;
1546.     warn $err_msg;
1547.     };
1548.
1549.     my $pending_delete_hash =
      $class->parse_pending_delete($conf->{pending}->{delete});
1550.     # FIXME: $force deletion is not implemented for CTs
1551.     foreach my $opt (sort keys %$pending_delete_hash) {
1552.     next if $selection && !$selection->{$opt};
1553.     eval {
1554.         if ($opt =~ m/^mp(\d+)$/) {
1555.         my $mp = $class->parse_volume($opt, $conf->{$opt});
1556.         if ($mp->{type} eq 'volume') {
1557.             $class->add_unused_volume($conf, $mp->{volume})
1558.             if !$class->is_volume_in_use($conf, $conf->{$opt}, 1,
      1);
1559.         }
1560.         } elsif ($opt =~ m/^unused(\d+)$/) {
1561.         PVE::LXC::delete_mountpoint_volume($storecfg, $vmid,
      $conf->{$opt})
1562.             if !$class->is_volume_in_use($conf, $conf->{$opt}, 1,
      1);
1563.         } elsif ($opt =~ m/^net(\d+)$/) {
1564.         if ($have_sdn) {
1565.             my $net = $class->parse_lxc_network($conf->{$opt});
1566.             eval {
      PVE::Network::SDN::Vnets::del_ips_from_mac($net->{bridge},
      $net->{hwaddr}, $conf->{hostname}) };
1567.             warn $@ if $@;
1568.         }
1569.         }
1570.     };
1571.     if (my $err = $@) {
1572.         $add_apply_error->($opt, $err);
1573.     } else {
```

```
1574.         delete $conf->{$opt};
1575.         $class->remove_from_pending_delete($conf, $opt);
1576.     }
1577.     }
1578.
1579.     $class->cleanup_pending($conf);
1580.
1581.     foreach my $opt (sort keys %{$conf->{pending}}) { # add/change
1582.     next if $opt eq 'delete'; # just to be sure
1583.     next if $selection && !$selection->{$opt};
1584.     eval {
1585.         if ($opt =~ m/^mp(\d+)$/) {
1586.         $class->apply_pending_mountpoint($vmid, $conf, $opt,
     $storecfg, 0);
1587.         } elsif ($opt =~ m/^net(\d+)$/) {
1588.         my $netid = $1;
1589.         my $net =
     $class->parse_lxc_network($conf->{pending}->{$opt});
1590.         $conf->{pending}->{$opt} =
     $class->print_lxc_network($net);
1591.         if ($have_sdn) {
1592.             if ($conf->{$opt}) {
1593.             my $old_net =
     $class->parse_lxc_network($conf->{$opt});
1594.             if ($old_net->{bridge} ne $net->{bridge} ||
     $old_net->{hwaddr} ne $net->{hwaddr}) {
1595.
     PVE::Network::SDN::Vnets::del_ips_from_mac($old_net->{bridge},
     $old_net->{hwaddr}, $conf->{name});
1596.
     PVE::Network::SDN::Vnets::add_next_free_cidr($net->{bridge},
     $conf->{hostname}, $net->{hwaddr}, $vmid, undef, 1);
1597.             }
1598.             } else {
1599.
     PVE::Network::SDN::Vnets::add_next_free_cidr($net->{bridge},
     $conf->{hostname}, $net->{hwaddr}, $vmid, undef, 1);
1600.             }
1601.         }
1602.         }
1603.     };
1604.     if (my $err = $@) {
1605.         $add_apply_error->($opt, $err);
1606.     } else {
1607.         $conf->{$opt} = delete $conf->{pending}->{$opt};
1608.     }
1609.     }
1610. }
1611.
1612. my $rescan_volume = sub {
1613.     my ($storecfg, $mp) = @_;
```

```
1614.     eval {
1615.     $mp->{size} = PVE::Storage::volume_size_info($storecfg,
      $mp->{volume}, 5);
1616.     };
1617.     warn "Could not rescan volume size - $@\n" if $@;
1618. };
1619.
1620. sub apply_pending_mountpoint {
1621.     my ($class, $vmid, $conf, $opt, $storecfg, $running) = @_;
1622.
1623.     my $mp = $class->parse_volume($opt, $conf->{pending}->{$opt});
1624.     my $old = $conf->{$opt};
1625.     if ($mp->{type} eq 'volume' && $mp->{volume} =~
      $PVE::LXC::NEW_DISK_RE) {
1626.     my $original_value = $conf->{pending}->{$opt};
1627.     my $vollist = PVE::LXC::create_disks(
1628.         $storecfg,
1629.         $vmid,
1630.         { $opt => $original_value },
1631.         $conf,
1632.         1,
1633.     );
1634.     if ($running) {
1635.         # Re-parse mount point:
1636.         my $mp = $class->parse_volume($opt,
      $conf->{pending}->{$opt});
1637.         eval {
1638.         PVE::LXC::mountpoint_hotplug($vmid, $conf, $opt, $mp,
      $storecfg);
1639.         };
1640.         my $err = $@;
1641.         if ($err) {
1642.         PVE::LXC::destroy_disks($storecfg, $vollist);
1643.         # The pending-changes code collects errors but keeps on
      looping through further
1644.         # pending changes, so unroll the change in $conf as well
      if destroy_disks()
1645.         # didn't die().
1646.         $conf->{pending}->{$opt} = $original_value;
1647.         die $err;
1648.         }
1649.     }
1650.     } else {
1651.     die "skip\n" if $running && defined($old); # TODO: "changing"
      mount points?
1652.     $rescan_volume->($storecfg, $mp) if $mp->{type} eq 'volume';
1653.     if ($running) {
1654.         PVE::LXC::mountpoint_hotplug($vmid, $conf, $opt, $mp,
      $storecfg);
1655.     }
1656.     $conf->{pending}->{$opt} = $class->print_ct_mountpoint($mp);
```

```
1657.     }
1658.
1659.     if (defined($old)) {
1660.     my $mp = $class->parse_volume($opt, $old);
1661.     if ($mp->{type} eq 'volume') {
1662.         $class->add_unused_volume($conf, $mp->{volume})
1663.         if !$class->is_volume_in_use($conf, $conf->{$opt}, 1, 1);
1664.     }
1665.     }
1666. }
1667.
1668. sub classify_mountpoint {
1669.     my ($class, $vol) = @_;
1670.     if ($vol =~ m!^/!) {
1671.     return 'device' if $vol =~ m!^/dev/!;
1672.     return 'bind';
1673.     }
1674.     return 'volume';
1675. }
1676.
1677. my $__is_volume_in_use = sub {
1678.     my ($class, $config, $volid) = @_;
1679.     my $used = 0;
1680.
1681.     $class->foreach_volume($config, sub {
1682.     my ($ms, $mountpoint) = @_;
1683.     return if $used;
1684.     $used = $mountpoint->{type} eq 'volume' &&
     $mountpoint->{volume} eq $volid;
1685.     });
1686.
1687.     return $used;
1688. };
1689.
1690. sub is_volume_in_use_by_snapshots {
1691.     my ($class, $config, $volid) = @_;
1692.
1693.     if (my $snapshots = $config->{snapshots}) {
1694.     foreach my $snap (keys %$snapshots) {
1695.         return 1 if $__is_volume_in_use->($class,
     $snapshots->{$snap}, $volid);
1696.     }
1697.     }
1698.
1699.     return 0;
1700. }
1701.
1702. sub is_volume_in_use {
1703.     my ($class, $config, $volid, $include_snapshots,
     $include_pending) = @_;
1704.     return 1 if $__is_volume_in_use->($class, $config, $volid);
```

```
1705.     return 1 if $include_snapshots &&
      $class->is_volume_in_use_by_snapshots($config, $volid);
1706.     return 1 if $include_pending && $__is_volume_in_use->($class,
      $config->{pending}, $volid);
1707.     return 0;
1708. }
1709.
1710. sub has_dev_console {
1711.     my ($class, $conf) = @_;
1712.
1713.     return !(defined($conf->{console}) && !$conf->{console});
1714. }
1715.
1716. sub has_lxc_entry {
1717.     my ($class, $conf, $keyname) = @_;
1718.
1719.     if (my $lxcconf = $conf->{lxc}) {
1720.     foreach my $entry (@$lxcconf) {
1721.         my ($key, undef) = @$entry;
1722.         return 1 if $key eq $keyname;
1723.     }
1724.     }
1725.
1726.     return 0;
1727. }
1728.
1729. sub get_tty_count {
1730.     my ($class, $conf) = @_;
1731.
1732.     return $conf->{tty} // $confdesc->{tty}->{default};
1733. }
1734.
1735. sub get_cmode {
1736.     my ($class, $conf) = @_;
1737.
1738.     return $conf->{cmode} // $confdesc->{cmode}->{default};
1739. }
1740.
1741. sub valid_volume_keys {
1742.     my ($class, $reverse) = @_;
1743.
1744.     my @names = ('rootfs');
1745.
1746.     for (my $i = 0; $i < $MAX_MOUNT_POINTS; $i++) {
1747.     push @names, "mp$i";
1748.     }
1749.
1750.     return $reverse ? reverse @names : @names;
1751. }
1752.
1753. sub valid_volume_keys_with_unused {
```

```
1754.     my ($class, $reverse) = @_;
1755.     my @names = $class->valid_volume_keys();
1756.     for (my $i = 0; $i < $MAX_UNUSED_DISKS; $i++) {
1757.     push @names, "unused$i";
1758.     }
1759.     return $reverse ? reverse @names : @names;
1760. }
1761.
1762. sub get_vm_volumes {
1763.     my ($class, $conf, $excludes) = @_;
1764.
1765.     my $vollist = [];
1766.
1767.     $class->foreach_volume($conf, sub {
1768.     my ($ms, $mountpoint) = @_;
1769.
1770.     return if $excludes && $ms eq $excludes;
1771.
1772.     my $volid = $mountpoint->{volume};
1773.     return if !$volid || $mountpoint->{type} ne 'volume';
1774.
1775.     my ($sid, $volname) = PVE::Storage::parse_volume_id($volid,
      1);
1776.     return if !$sid;
1777.
1778.     push @$vollist, $volid;
1779.     });
1780.
1781.     return $vollist;
1782. }
1783.
1784. sub get_replicatable_volumes {
1785.     my ($class, $storecfg, $vmid, $conf, $cleanup, $noerr) = @_;
1786.
1787.     my $volhash = {};
1788.
1789.     my $test_volid = sub {
1790.     my ($volid, $mountpoint) = @_;
1791.
1792.     return if !$volid;
1793.
1794.     my $mptype = $mountpoint->{type};
1795.     my $replicate = $mountpoint->{replicate} // 1;
1796.
1797.     if ($mptype ne 'volume') {
1798.         # skip bindmounts if replicate = 0 even for cleanup,
1799.         # since bind mounts could not have been replicated ever
1800.         return if !$replicate;
1801.         die "unable to replicate mountpoint type '$mptype'\n";
1802.     }
1803.
```

```
1804.     my ($storeid, $volname) =
      PVE::Storage::parse_volume_id($volid, $noerr);
1805.     return if !$storeid;
1806.
1807.     my $scfg = PVE::Storage::storage_config($storecfg, $storeid);
1808.     return if $scfg->{shared};
1809.
1810.     my ($path, $owner, $vtype) = PVE::Storage::path($storecfg,
      $volid);
1811.     return if !$owner || ($owner != $vmid);
1812.
1813.     die "unable to replicate volume '$volid', type '$vtype'\n" if
      $vtype ne 'images';
1814.
1815.     return if !$cleanup && !$replicate;
1816.
1817.     if (!PVE::Storage::volume_has_feature($storecfg, 'replicate',
      $volid)) {
1818.         return if $cleanup || $noerr;
1819.         die "missing replicate feature on volume '$volid'\n";
1820.     }
1821.
1822.     $volhash->{$volid} = 1;
1823.     };
1824.
1825.     $class->foreach_volume($conf, sub {
1826.     my ($ms, $mountpoint) = @_;
1827.     $test_volid->($mountpoint->{volume}, $mountpoint);
1828.     });
1829.
1830.     foreach my $snapname (keys %{$conf->{snapshots}}) {
1831.     my $snap = $conf->{snapshots}->{$snapname};
1832.     $class->foreach_volume($snap, sub {
1833.         my ($ms, $mountpoint) = @_;
1834.         $test_volid->($mountpoint->{volume}, $mountpoint);
1835.         });
1836.     }
1837.
1838.     # add 'unusedX' volumes to volhash
1839.     foreach my $key (keys %$conf) {
1840.     if ($key =~ m/^unused/) {
1841.         $test_volid->($conf->{$key}, { type => 'volume', replicate
      => 1 });
1842.     }
1843.     }
1844.
1845.     return $volhash;
1846. }
1847.
1848. sub get_backup_volumes {
1849.     my ($class, $conf) = @_;
```

```
1850.
1851.     my $return_volumes = [];
1852.
1853.     my $test_mountpoint = sub {
1854.     my ($key, $volume) = @_;
1855.
1856.     my ($included, $reason) =
     $class->mountpoint_backup_enabled($key, $volume);
1857.
1858.     push @$return_volumes, {
1859.         key => $key,
1860.         included => $included,
1861.         reason => $reason,
1862.         volume_config => $volume,
1863.     };
1864.     };
1865.
1866.     PVE::LXC::Config->foreach_volume($conf, $test_mountpoint);
1867.
1868.     return $return_volumes;
1869. }
1870.
1871. sub get_derived_property {
1872.     my ($class, $conf, $name) = @_;
1873.
1874.     if ($name eq 'max-cpu') {
1875.     return $conf->{cpulimit} || $conf->{cores} || 0;
1876.     } elsif ($name eq 'max-memory') {
1877.     return ($conf->{memory} || 512) * 1024 * 1024;
1878.     } else {
1879.     die "unknown derived property - $name\n";
1880.     }
1881. }
1882.
1883. sub foreach_passthrough_device {
1884.     my ($class, $conf, $func, @param) = @_;
1885.
1886.     for my $key (keys %$conf) {
1887.     next if $key !~ m/^dev(\d+)$/;
1888.
1889.     my $device = $class->parse_device($conf->{$key});
1890.
1891.     $func->($key, $device, @param);
1892.     }
1893. }
1894.
1895. 1;
```

In der Datei **/usr/share/perl5/PVE/LXC/Setup.pm** muss eisfair auch noch an einigen Stellen eingefügt werden.

Die entsprechenden Stellen sind farblich hervorgehoben, (Zeilen 16, 31, 74-75)

/usr/share/perl5/PVE/LXC/Setup.pm

```
 1. package PVE::LXC::Setup;
 2.
 3. use strict;
 4. use warnings;
 5.
 6. use POSIX;
 7. use Cwd 'abs_path';
 8.
 9. use PVE::Tools;
10.
11. use PVE::LXC::Setup::Alpine;
12. use PVE::LXC::Setup::ArchLinux;
13. use PVE::LXC::Setup::CentOS;
14. use PVE::LXC::Setup::Debian;
15. use PVE::LXC::Setup::Devuan;
16. use PVE::LXC::Setup::Eisfair;
17. use PVE::LXC::Setup::Fedora;
18. use PVE::LXC::Setup::Gentoo;
19. use PVE::LXC::Setup::SUSE;
20. use PVE::LXC::Setup::Ubuntu;
21. use PVE::LXC::Setup::NixOS;
22. use PVE::LXC::Setup::OpenEuler;
23. use PVE::LXC::Setup::Unmanaged;
24.
25. my $plugins = {
26.     alpine    => 'PVE::LXC::Setup::Alpine',
27.     archlinux => 'PVE::LXC::Setup::ArchLinux',
28.     centos    => 'PVE::LXC::Setup::CentOS',
29.     debian    => 'PVE::LXC::Setup::Debian',
30.     devuan    => 'PVE::LXC::Setup::Devuan',
31.     eisfair   => 'PVE::LXC::Setup::Eisfair',
32.     fedora    => 'PVE::LXC::Setup::Fedora',
33.     gentoo    => 'PVE::LXC::Setup::Gentoo',
34.     openeuler => 'PVE::LXC::Setup::OpenEuler',
35.     opensuse  => 'PVE::LXC::Setup::SUSE',
36.     ubuntu    => 'PVE::LXC::Setup::Ubuntu',
37.     nixos     => 'PVE::LXC::Setup::NixOS',
38.     unmanaged => 'PVE::LXC::Setup::Unmanaged',
39. };
40.
```

```
41. # a map to allow supporting related distro flavours
42. my $plugin_alias = {
43.     'opensuse-leap' => 'opensuse',
44.     'opensuse-tumbleweed' => 'opensuse',
45.     'opensuse-slowroll' => 'opensuse',
46.     'openEuler' => 'openeuler',
47.     arch => 'archlinux',
48.     sles => 'opensuse',
49. };
50.
51. my $autodetect_type = sub {
52.     my ($self, $rootdir, $os_release) = @_;
53.
54.     if (my $id = $os_release->{ID}) {
55.         return $id if $plugins->{$id};
56.         return $plugin_alias->{$id} if $plugin_alias->{$id};
57.         warn "unknown ID '$id' in /etc/os-release file, trying
    fallback detection\n";
58.     }
59.
60.     # fallback compatibility checks
61.
62.     my $lsb_fn = "$rootdir/etc/lsb-release";
63.     if (-f $lsb_fn) {
64.         my $data =  PVE::Tools::file_get_contents($lsb_fn);
65.         if ($data =~ m/^DISTRIB_ID=Ubuntu$/im) {
66.             return 'ubuntu';
67.         }
68.     }
69.
70.     if (-f "$rootdir/etc/debian_version") {
71.         return "debian";
72.     } elsif (-f "$rootdir/etc/devuan_version") {
73.         return "devuan";
74.     } elsif (-f "$rootdir/etc/version") {
75.         return "eisfair";
76.     } elsif (-f  "$rootdir/etc/SuSE-brand" || -f
    "$rootdir/etc/SuSE-release") {
77.         return "opensuse";
78.     } elsif (-f  "$rootdir/etc/fedora-release") {
79.         return "fedora";
80.     } elsif (-f  "$rootdir/etc/centos-release" || -f
    "$rootdir/etc/redhat-release") {
81.         return "centos";
82.     } elsif (-f  "$rootdir/etc/arch-release") {
83.         return "archlinux";
84.     } elsif (-f  "$rootdir/etc/alpine-release") {
85.         return "alpine";
86.     } elsif (-f  "$rootdir/etc/gentoo-release") {
87.         return "gentoo";
88.     } elsif (-d  "$rootdir/nix/store") {
```

```
89.         return "nixos";
90.     } elsif (-f "$rootdir/etc/openEuler-release") {
91.         return "openeuler";
92.     } elsif (-f "$rootdir/etc/os-release") {
93.         die "unable to detect OS distribution\n";
94.     } else {
95.         warn "/etc/os-release file not found and autodetection
    failed, falling back to 'unmanaged'\n";
96.         return "unmanaged";
97.     }
98. };
99.
100. sub new {
101.     my ($class, $conf, $rootdir, $type) = @_;
102.
103.     die "no root directory\n" if !$rootdir || $rootdir eq '/';
104.
105.     my $self = bless { conf => $conf, rootdir => $rootdir},
    $class;
106.
107.     my $os_release = $self->get_ct_os_release();
108.
109.     if ($conf->{ostype} && $conf->{ostype} eq 'unmanaged') {
110.         $type = 'unmanaged';
111.     } elsif (!defined($type)) {
112.         # try to autodetect type
113.         $type = &$autodetect_type($self, $rootdir, $os_release);
114.         my $expected_type = $conf->{ostype} || $type;
115.
116.         if ($type ne $expected_type) {
117.             warn "WARNING: /etc not present in CT, is the rootfs
    mounted?\n"
118.                 if ! -e "$rootdir/etc";
119.             warn "got unexpected ostype ($type !=
    $expected_type)\n"
120.         }
121.     }
122.
123.     my $plugin_class = $plugins->{$type} || die "no such OS type
    '$type'\n";
124.
125.     my $plugin = $plugin_class->new($conf, $rootdir, $os_release);
126.     $self->{plugin} = $plugin;
127.     $self->{in_chroot} = 0;
128.
129.     # Cache some host files we need access to:
130.     $plugin->{host_resolv_conf} =
    PVE::INotify::read_file('resolvconf');
131.     $plugin->{host_timezone} =
    PVE::INotify::read_file('timezone');
132.
```

```
133.     abs_path('/etc/localtime') =~ m|^(/.+)| or die "invalid
    /etc/localtime\n"; # untaint
134.     $plugin->{host_localtime} = $1;
135.
136.     # pass on user namespace information:
137.     my ($id_map, $root_uid, $root_gid) =
    PVE::LXC::parse_id_maps($conf);
138.     if (@$id_map) {
139.         $plugin->{id_map} = $id_map;
140.         $plugin->{root_uid} = $root_uid;
141.         $plugin->{root_gid} = $root_gid;
142.     }
143.
144.     # if arch is unset, we try to autodetect it
145.     if (!defined($conf->{arch})) {
146.         my $arch = eval { $self->protected_call(sub {
    $plugin->detect_architecture() }) };
147.
148.         if (my $err = $@) {
149.             warn "Architecture detection failed: $err" if $err;
150.         }
151.
152.         if (!defined($arch)) {
153.             $arch = 'amd64';
154.             print "Falling back to $arch.\nUse `pct set VMID --
    arch ARCH` to change.\n";
155.         } else {
156.             print "Detected container architecture: $arch\n";
157.         }
158.
159.         $conf->{arch} = $arch;
160.     }
161.
162.     return $self;
163. }
164.
165. # Forks into a chroot and executes $sub
166. sub protected_call {
167.     my ($self, $sub) = @_;
168.
169.     # avoid recursion:
170.     return $sub->() if $self->{in_chroot};
171.
172.     pipe(my $res_in, my $res_out) or die "pipe failed: $!\n";
173.
174.     my $child = fork();
175.     die "fork failed: $!\n" if !defined($child);
176.
177.     if (!$child) {
178.         close($res_in);
179.         # avoid recursive forks
```

```
180.         $self->{in_chroot} = 1;
181.         eval {
182.             my $rootdir = $self->{rootdir};
183.             chroot($rootdir) or die "failed to change root to:
    $rootdir: $!\n";
184.             chdir('/') or die "failed to change to root
    directory\n";
185.             my $res = $sub->();
186.             if (defined($res)) {
187.                 print {$res_out} "$res";
188.                 $res_out->flush();
189.             }
190.         };
191.         if (my $err = $@) {
192.             warn $err;
193.             POSIX::_exit(1);
194.         }
195.         POSIX::_exit(0);
196.     }
197.     close($res_out);
198.     my $result = do { local $/ = undef; <$res_in> };
199.     while (waitpid($child, 0) != $child) {}
200.     if ($? != 0) {
201.         my $method = (caller(1))[3];
202.         die "error in setup task $method\n";
203.     }
204.     return $result;
205. }
206.
207. sub template_fixup {
208.     my ($self) = @_;
209.     $self->protected_call(sub {
    $self->{plugin}->template_fixup($self->{conf}) });
210. }
211.
212. sub setup_network {
213.     my ($self) = @_;
214.     $self->protected_call(sub {
    $self->{plugin}->setup_network($self->{conf}) });
215. }
216.
217. sub set_hostname {
218.     my ($self) = @_;
219.     $self->protected_call(sub {
    $self->{plugin}->set_hostname($self->{conf}) });
220. }
221.
222. sub set_dns {
223.     my ($self) = @_;
224.     $self->protected_call(sub {
    $self->{plugin}->set_dns($self->{conf}) });
```

```
225. }
226.
227. sub set_timezone {
228.     my ($self) = @_;
229.     $self->protected_call(sub {
    $self->{plugin}->set_timezone($self->{conf}) });
230. }
231.
232. sub setup_init {
233.     my ($self) = @_;
234.     $self->protected_call(sub {
    $self->{plugin}->setup_init($self->{conf}) });
235. }
236.
237. sub set_user_password {
238.     my ($self, $user, $pw) = @_;
239.     $self->protected_call(sub {
    $self->{plugin}->set_user_password($self->{conf}, $user, $pw) });
240. }
241.
242. my sub generate_ssh_key { # create temporary key in hosts' /run,
    then read and unlink
243.     my ($type, $comment) = @_;
244.
245.     my $key_id = '';
246.     my $keygen_outfunc = sub {
247.         if ($_[0] =~ m/^((?:[0-9a-f]{2}:)+[0-9a-
    f]{2}|SHA256:[0-9a-z+\/]{43})\s+\Q$comment\E$/i) {
248.             $key_id = $_[0];
249.         }
250.     };
251.     my $file = "/run/pve/.tmp$$.$type";
252.     PVE::Tools::run_command(
253.         ['ssh-keygen', '-f', $file, '-t', $type, '-N', '', '-E',
    'sha256', '-C', $comment],
254.         outfunc => $keygen_outfunc,
255.     );
256.     my ($private) = (PVE::Tools::file_get_contents($file) =~
    /^(.*)$/sg); # untaint
257.     my ($public) = (PVE::Tools::file_get_contents("$file.pub") =~
    /^(.*)$/sg); # untaint
258.     unlink $file, "$file.pub";
259.
260.     return ($key_id, $private, $public);
261. }
262.
263. sub rewrite_ssh_host_keys {
264.     my ($self) = @_;
265.
266.     my $plugin = $self->{plugin};
267.
```

```
268.     my $keynames = $plugin->ssh_host_key_types_to_generate();
269.
270.     return if ! -d "$self->{rootdir}/etc/ssh" || !$keynames ||
     !scalar(keys $keynames->%*);
271.
272.     my $hostname = $self->{conf}->{hostname} || 'localhost';
273.     $hostname =~ s/\..*$//;
274.
275.     my $keyfiles = [];
276.     for my $keytype (keys $keynames->%*) {
277.         my $basename = $keynames->{$keytype};
278.         print "Creating SSH host key '$basename' - this may take
     some time ...\n";
279.         my ($id, $private, $public) = generate_ssh_key($keytype,
     "root\@$hostname");
280.         print "done: $id\n";
281.
282.         push $keyfiles->@*, ["/etc/ssh/$basename", $private,
     0600], ["/etc/ssh/$basename.pub", $public, 0644];
283.     }
284.
285.     $self->protected_call(sub { # write them now all to the CTs
     rootfs at once
286.         for my $file ($keyfiles->@*) {
287.             $plugin->ct_file_set_contents($file->@*);
288.         }
289.     });
290. }
291.
292. sub pre_start_hook {
293.     my ($self) = @_;
294.
295.     $self->protected_call(sub {
     $self->{plugin}->pre_start_hook($self->{conf}) });
296. }
297.
298. sub post_clone_hook {
299.     my ($self, $conf) = @_;
300.
301.     $self->protected_call(sub {
     $self->{plugin}->post_clone_hook($conf) });
302. }
303.
304. sub post_create_hook {
305.     my ($self, $root_password, $ssh_keys) = @_;
306.
307.     $self->protected_call(sub {
308.         $self->{plugin}->post_create_hook($self->{conf},
     $root_password, $ssh_keys);
309.     });
310.     $self->rewrite_ssh_host_keys();
```

```
311. }
312.
313. sub unified_cgroupv2_support {
314.     my ($self) = @_;
315.
316.     return
     $self->{plugin}->unified_cgroupv2_support($self->get_ct_init_path(
     ));
317. }
318.
319. # os-release(5):
320. #   (...) a newline-separated list of environment-like shell-
     compatible
321. #   variable assignments. (...) beyond mere variable assignments,
     no shell
322. #   features are supported (this means variable expansion is
     explicitly not
323. #   supported) (...). Variable assignment values must be enclosed
     in double or
324. #   single quotes *if* they include spaces, semicolons or other
     special
325. #   characters outside of A-Z, a-z, 0-9. Shell special characters
     ("$", quotes,
326. #   backslash, backtick) must be escaped with backslashes (...).
     All strings
327. #   should be in UTF-8 format, and non-printable characters should
     not be used.
328. #   It is not supported to concatenate multiple individually
     quoted strings.
329. #   Lines beginning with "#" shall be ignored as comments.
330. my $parse_os_release = sub {
331.     my ($data) = @_;
332.     my $variables = {};
333.     while (defined($data) && $data =~ /^(.+)$/gm) {
334.         next if $1 !~ /^\s*([a-zA-Z_][a-zA-Z0-9_]*)=(.*)$/;
335.         my ($var, $content) = ($1, $2);
336.         chomp $content;
337.
338.         if ($content =~ /^'([^']*)'/) {
339.             $variables->{$var} = $1;
340.         } elsif ($content =~ /^"((?:[^"\\]|\\.)*)"/) {
341.             my $s = $1;
342.             $s =~ s/(\\["'`nt\$\\])/"\"$1\""/eeg;
343.             $variables->{$var} = $s;
344.         } elsif ($content =~ /^([A-Za-z0-9]*)/) {
345.             $variables->{$var} = $1;
346.         }
347.     }
348.     return $variables;
349. };
350.
```

```
351. sub get_ct_os_release {
352.     my ($self) = @_;
353.
354.     my $data = $self->protected_call(sub {
355.         if (-f '/etc/os-release') {
356.             return PVE::Tools::file_get_contents('/etc/os-
    release');
357.         } elsif (-f '/usr/lib/os-release') {
358.             return PVE::Tools::file_get_contents('/usr/lib/os-
    release');
359.         }
360.         return undef;
361.     });
362.
363.     return &$parse_os_release($data);
364. }
365.
366. # Checks whether /sbin/init is a symlink, and if it is, resolves
    it to the actual binary
367. sub get_ct_init_path {
368.     my ($self) = @_;
369.
370.     my $init = $self->protected_call(sub {
371.         return $self->{plugin}->get_ct_init_path();
372.     });
373.
374.     return $init;
375. }
376.
377. 1;
```

Fügt nach Zeile 15 (**use PVE::LXC::Setup::Devuan;**) bitte noch

```
use PVE::LXC::Setup::Eisfair;
```

etwas weiter unten bei **my $plugins = {** dann noch

```
eisfair => ‚PVE::LXC::SETUP::Eisfair‘;
```

und nochmals weiter unten nach **return „devuan**“; suchen und darunter dann

```
} elsif (-f „$rootdir/etc/version“) {
    return „eisfair“;
```

einfügen.

Jetzt noch das file **/usr/share/perl5/PVR/LXC/Setup/Eisfair.pm** anlegen und mit diesem Inhalt füllen:

/usr/share/perl5/PVR/LXC/Setup/Eisfair.pm

```
package PVE::LXC::Setup::Eisfair;

use strict;
use warnings;

use PVE::LXC;
use PVE::Network;

use base qw(PVE::LXC::Setup::Base);

sub new {
    my ($class, $conf, $rootdir) = @_;

    my $self = { conf => $conf, rootdir => $rootdir };

    $conf->{ostype} = "eisfair";

    return bless $self, $class;
}

# Eisfair verwendet systemd, daher benötigen wir kein spezielles
/dev/lxc-Verzeichnis.
sub devttydir {
    return '';
}

sub setup_init {
    my ($self, $conf) = @_;

    # Überprüfen, ob systemd verwendet wird
    my $systemd = $self->ct_readlink('/sbin/init');
    if (defined($systemd) && $systemd =~ m@/systemd$@) {
        $self->setup_container_getty_service($conf);

        # Vorkonfiguration für systemd-basierte Dienste
        $self->setup_systemd_preset({
            'NetworkManager.service' => 0,
        });
    }
}

sub setup_network {
    my ($self, $conf) = @_;

    my $base_filename = "/etc/config.d/base";
    my $dhcpc_filename = "/etc/config.d/dhcpc";

    # Alte Base-Konfigurationsdatei einlesen
    open(my $base_fh, '<', $base_filename) or die "Kann $base_filename
```

```
nicht öffnen: $!";
    my @base_lines = <$base_fh>;
    close($base_fh);

    # Initialisiere Standard-Netzwerk-Konfiguration
    my $network_settings = {
        address => '',
        netmask => '',
        gateway => $conf->{ip_default_gateway} || '',
        macaddr => $conf->{hwaddr} || '',
        network => '',
        eth_name => 'eth0',
        dhcp    => 0,
    };

    # Netzwerkschnittstellen verarbeiten
    foreach my $k (keys %$conf) {
        next if $k !~ m/^net(\d+)$/;  # Nur Schnittstellen mit "netX"
bearbeiten
        my $ind = $1;                 # Netzwerkindex extrahieren
        my $d = PVE::LXC::Config->parse_lxc_network($conf->{$k});

        # DHCP prüfen
        if ($d->{ip} && $d->{ip} eq 'dhcp') {
            $network_settings->{dhcp} = 1;
            $network_settings->{eth_name} = "eth$ind";
            next;
        }

        # Statische IP und Netzmaske extrahieren
        if (defined($d->{ip})) {
            my $ipinfo = PVE::LXC::parse_ipv4_cidr($d->{ip});
            $network_settings->{address} = $ipinfo->{address};
            $network_settings->{netmask} = $ipinfo->{netmask};
            $network_settings->{network} =
$self->get_network_address($d->{ip});
        }

        # Gateway und MAC-Adresse
        $network_settings->{gateway} = $d->{gw} if $d->{gw};
        $network_settings->{macaddr} = $d->{hwaddr} || '';
        $network_settings->{eth_name} = "eth$ind";
    }

    # Hostname aus der Konfiguration holen
    my $hostname = $conf->{hostname} || 'eisfair';

    # DNS- und Domain-Einstellungen abrufen
    my ($searchdomain, $nameserver) = $self->lookup_dns_conf($conf);

    if ($network_settings->{dhcp}) {
```

```
        # DHCP-Konfigurationsdatei bearbeiten
        open(my $dhcpc_fh, '<', $dhcpc_filename) or die "Kann
$dhcpc_filename nicht öffnen: $!";
        my @dhcpc_lines = <$dhcpc_fh>;
        close($dhcpc_fh);

        foreach my $line (@dhcpc_lines) {
            if ($line =~ /^START_DHCPC=/) {
                $line = "START_DHCPC='yes'\n";
            } elsif ($line =~ /^DHCP_HOSTNAME=/) {
                $line = "DHCP_HOSTNAME='$hostname'\n";
            }
        }

        # DHCP-Konfiguration speichern
        open($dhcpc_fh, '>', $dhcpc_filename) or die "Kann
$dhcpc_filename nicht schreiben: $!";
        print $dhcpc_fh @dhcpc_lines;
        close($dhcpc_fh);

        # Ausführen des dhcpc.sh-Skripts
        $self->start_meinscript('dhcpc.sh');

        # Dienste starten
        $self->start_service('resolver');
        $self->start_service('dhcpc');
    } else {
        # Statische Netzwerkkonfiguration
        foreach my $line (@base_lines) {
            if ($line =~ /^HOSTNAME=/) {
                $line = "HOSTNAME='$hostname'\n";
            } elsif ($line =~ /^IP_ETH_1_IPADDR=/) {
                $line =
"IP_ETH_1_IPADDR='$network_settings->{address}'\n";
            } elsif ($line =~ /^IP_ETH_1_NETMASK=/) {
                $line =
"IP_ETH_1_NETMASK='$network_settings->{netmask}'\n";
            } elsif ($line =~ /^IP_ETH_1_MACADDR=/) {
                $line =
"IP_ETH_1_MACADDR='$network_settings->{macaddr}'\n";
            } elsif ($line =~ /^IP_DEFAULT_GATEWAY=/) {
                $line =
"IP_DEFAULT_GATEWAY='$network_settings->{gateway}'\n";
            } elsif ($line =~ /^IP_ETH_1_NAME=/) {
                $line =
"IP_ETH_1_NAME='$network_settings->{eth_name}'\n";
            } elsif ($line =~ /^IP_ETH_1_NETWORK=/) {
                $line =
"IP_ETH_1_NETWORK='$network_settings->{network}'\n";
            } elsif ($line =~ /^DOMAIN_NAME=/) {
                $line = "DOMAIN_NAME='$searchdomain'\n";
```

```
            } elsif ($line =~ /^DNS_SERVER=/) {
                $line = "DNS_SERVER='$nameserver'\n";
            }
        }

        # Base-Konfiguration speichern
        open($base_fh, '>', $base_filename) or die "Kann $base_filename
nicht schreiben: $!";
        print $base_fh @base_lines;
        close($base_fh);

        # DHCP-Konfigurationsdatei bearbeiten
        open(my $dhcpc_fh, '<', $dhcpc_filename) or die "Kann
$dhcpc_filename nicht öffnen: $!";
        my @dhcpc_lines = <$dhcpc_fh>;
        close($dhcpc_fh);

        foreach my $line (@dhcpc_lines) {
            if ($line =~ /^START_DHCPC=/) {
                $line = "START_DHCPC='no'\n";
            }
        }

        # DHCP-Konfiguration speichern
        open($dhcpc_fh, '>', $dhcpc_filename) or die "Kann
$dhcpc_filename nicht schreiben: $!";
        print $dhcpc_fh @dhcpc_lines;
        close($dhcpc_fh);

        # Ausführen des dhcpc.sh-Skripts
        $self->start_meinscript('dhcpc.sh');


        # Dienste verwalten
        $self->stop_service('dhcpc');
        $self->restart_service('resolver');
    }

    print "Netzwerk-, Hostname-, Domain- und DNS-Konfiguration für
Eisfair wurde aktualisiert.\n";
}

# Script ausführen
sub start_meinscript {
    my ($self, $scriptname) = @_;
    system("/bin/sh /var/install/config.d/$scriptname") == 0 or die
"Konnte Script $scriptname nicht starten: $!";
}

# Dienstverwaltungsfunktionen bleiben unverändert
sub start_service {
```

```
    my ($self, $service) = @_;
    system("/usr/bin/systemctl start $service") == 0 or die "Konnte
Dienst $service nicht starten: $!";
}

sub restart_service {
    my ($self, $service) = @_;
    system("/usr/bin/systemctl restart $service") == 0 or die "Konnte
Dienst $service nicht neu starten: $!";
}

sub stop_service {
    my ($self, $service) = @_;
    system("/usr/bin/systemctl stop $service") == 0 or die "Konnte
Dienst $service nicht stoppen: $!";
}

# Funktionalität für Netzwerkadressberechnungen bleibt unverändert
sub get_network_address {
    my ($self, $ip_cidr) = @_;

    my $ipinfo = PVE::LXC::parse_ipv4_cidr($ip_cidr);
    my $ip_bin = $self->ip_to_binary($ipinfo->{address});
    my $netmask_bin = $self->ip_to_binary($ipinfo->{netmask});
    my $network_bin = $self->bitwise_and($ip_bin, $netmask_bin);
    return $self->binary_to_ip($network_bin);
}

sub ip_to_binary {
    my ($self, $ip) = @_;
    return join('', map { sprintf("%08b", $_) } split(/\./, $ip));
}

sub binary_to_ip {
    my ($self, $binary) = @_;
    return join('.', map { oct("0b$_") } ($binary =~ /(.{8})/g));
}

sub bitwise_and {
    my ($self, $bin1, $bin2) = @_;
    return join('', map { substr($bin1, $_, 1) & substr($bin2, $_, 1) }
0 .. length($bin1) - 1);
}

1;
```

Danach am besten einmal den Proxmox Host neustarten oder zumindest die PVE Dienste:

```
systemctl restart pveproxy
```

```
systemctl restart pvedaemon
```

Jetzt sollte eigentlich ein Eisfair LXC Container erstellt werden können. :)

Viel Spaß
Christian Richter, richterch@gmx.de