Boot Configuration

Author: Masami Hiramatsu <mhiramat@kernel.org>

Overview

The boot configuration expands the current kernel command line to support additional key-value data when booting the kernel in an efficient way. This allows administrators to pass a structured-Key config file.

Config File Syntax

The boot config syntax is a simple structured key-value. Each key consists of dot-connected-words, and key and value are connected by =. The value has to be terminated by semi-colon (;) or newline (\n). For array value, array entries are separated by comma (,).

```
KEY[.WORD[...]] = VALUE[, VALUE2[...]][;]
```

Unlike the kernel command line syntax, spaces are OK around the comma and =.

Each key word must contain only alphabets, numbers, dash (-) or underscore (_). And each value only contains printable characters or spaces except for delimiters such as semi-colon (;), new-line (\n), comma (,), hash (#) and closing brace (}).

If you want to use those delimiters in a value, you can use either double- quotes ("VALUE") or single-quotes ('VALUE') to quote it. Note that you can not escape these quotes.

There can be a key which doesn't have value or has an empty value. Those keys are used for checking if the key exists or not (like a boolean).

Key-Value Syntax

The boot config file syntax allows user to merge partially same word keys by brace. For example:

```
foo.bar.baz = value1
foo.bar.qux.quux = value2
```

These can be written also in:

```
foo.bar {
   baz = value1
   qux.quux = value2
}
```

Or more shorter, written as following:

```
foo.bar { baz = value1; qux.quux = value2 }
```

In both styles, same key words are automatically merged when parsing it at boot time. So you can append similar trees or key-values.

Same-key Values

It is prohibited that two or more values or arrays share a same-key. For example,:

```
foo = bar, baz
foo = qux  # !ERROR! we can not re-define same key
```

If you want to update the value, you must use the override operator := explicitly. For example:

```
foo = bar, baz
foo := qux
```

then, the qux is assigned to foo key. This is useful for overriding the default value by adding (partial) custom bootconfigs without parsing the default bootconfig.

If you want to append the value to existing key as an array member, you can use += operator. For example:

```
foo = bar, baz
foo += qux
```

In this case, the key foo has bar, baz and qux.

Moreover, sub-keys and a value can coexist under a parent key. For example, following config is allowed.:

```
foo = value1
foo.bar = value2
```

```
foo := value3 # This will update foo's value.
```

Note, since there is no syntax to put a raw value directly under a structured key, you have to define it outside of the brace. For example:

```
foo {
    bar = value1
    bar {
        baz = value2
        qux = value3
    }
}
```

Also, the order of the value node under a key is fixed. If there are a value and subkeys, the value is always the first child node of the key. Thus if user specifies subkeys first, e.g.:

```
foo.bar = value1
foo = value2
```

In the program (and /proc/bootconfig), it will be shown as below:

```
foo = value2
foo.bar = value1
```

Comments

The config syntax accepts shell-script style comments. The comments starting with hash ("#") until newline ("n") will be ignored.

```
# comment line
foo = value # value is set to foo.
bar = 1, # 1st element
      2, # 2nd element
      3  # 3rd element
```

This is parsed as below:

```
foo = value
bar = 1, 2, 3
```

Note that you can not put a comment between value and delimiter(`,` or `;`). This means following config has a syntax error

```
key = 1 # comment
      ,2
```

/proc/bootconfig

/proc/bootconfig is a user-space interface of the boot config. Unlike /proc/cmdline, this file shows the key-value style list. Each key-value pair is shown in each line with following style:

```
KEY[.WORDS...] = "[VALUE]"[,"VALUE2"...]
```

Boot Kernel With a Boot Config

Since the boot configuration file is loaded with initrd, it will be added to the end of the initrd (initramfs) image file with padding, size, checksum and 12-byte magic word as below.

[initrd][bootconfig][padding][size(le32)][checksum(le32)][#BOOTCONFIGn]

The size and checksum fields are unsigned 32bit little endian value.

When the boot configuration is added to the initrd image, the total file size is aligned to 4 bytes. To fill the gap, null characters (`\0`) will be added. Thus the `size` is the length of the bootconfig file + padding bytes.

The Linux kernel decodes the last part of the initrd image in memory to get the boot configuration data. Because of this "piggyback" method, there is no need to change or update the boot loader and the kernel image itself as long as the boot loader passes the correct initrd file size. If by any chance, the boot loader passes a longer size, the kernel fails to find the bootconfig data.

To do this operation, Linux kernel provides `bootconfig` command under tools/bootconfig, which allows admin to apply or delete the config file to/from initrd image. You can build it by the following command:

```
# make -C tools/bootconfig
```

To add your boot config file to initrd image, run bootconfig as below (Old data is removed automatically if exists):

```
# tools/bootconfig/bootconfig -a your-config /boot/initrd.img-X.Y.Z
```

To remove the config from the image, you can use -d option as below:

```
# tools/bootconfig/bootconfig -d /boot/initrd.img-X.Y.Z
```

Then add "bootconfig" on the normal kernel command line to tell the kernel to look for the bootconfig at the end of the initrd file.

Kernel parameters via Boot Config

In addition to the kernel command line, the boot config can be used for passing the kernel parameters. All the key-value pairs under `kernel` key will be passed to kernel cmdline directly. Moreover, the key-value pairs under `init` will be passed to init process via the cmdline. The parameters are concatinated with user-given kernel cmdline string as the following order, so that the command line parameter can override bootconfig parameters (this depends on how the subsystem handles parameters but in general, earlier parameter will be overwritten by later one.):

```
[bootconfig params][cmdline params] -- [bootconfig init params][cmdline init params]
```

Here is an example of the bootconfig file for kernel/init parameters.:

```
kernel {
  root = 01234567-89ab-cdef-0123-456789abcd
}
init {
 splash
}
```

This will be copied into the kernel cmdline string as the following:

```
root="01234567-89ab-cdef-0123-456789abcd" -- splash
```

If user gives some other command line like,:

```
ro bootconfig -- quiet
```

The final kernel cmdline will be the following:

```
root="01234567-89ab-cdef-0123-456789abcd" ro bootconfig -- splash quiet
```

Config File Limitation

Currently the maximum config size size is 32KB and the total key-words (not key-value entries) must be under 1024 nodes. Note: this is not the number of entries but nodes, an entry must consume more than 2 nodes (a key-word and a value). So theoretically, it will be up to 512 key-value pairs. If keys contains 3 words in average, it can contain 256 key-value pairs. In most cases, the number of config items will be under 100 entries and smaller than 8KB, so it would be enough. If the node number exceeds 1024, parser returns an error even if the file size is smaller than 32KB. (Note that this maximum size is not including the padding null characters.) Anyway, since bootconfig command verifies it when appending a boot config to initrd image, user can notice it before boot.

Bootconfig APIs

User can query or loop on key-value pairs, also it is possible to find a root (prefix) key node and find key-values under that node.

If you have a key string, you can query the value directly with the key using xbc_find_value(). If you want to know what keys exist in the boot config, you can use xbc_for_each_key_value() to iterate key-value pairs. Note that you need to use xbc_array_for_each_value() for accessing each array's value, e.g.:

```
vnode = NULL;
xbc_find_value("key.word", &vnode);
if (vnode && xbc_node_is_array(vnode))
   xbc_array_for_each_value(vnode, value) {
     printk("%s ", value);
   }
```

If you want to focus on keys which have a prefix string, you can use xbc_find_node() to find a node by the prefix string, and iterate keys under the prefix node with xbc_node_for_each_key_value().

But the most typical usage is to get the named value under prefix or get the named array under prefix as below:

```
root = xbc_find_node("key.prefix");
value = xbc_node_find_value(root, "option", &vnode);
...
xbc_node_for_each_array_value(root, "array-option", value, anode) {
   ...
}
```

This accesses a value of "key.prefix.option" and an array of "key.prefix.array-option".

Locking is not needed, since after initialization, the config becomes read-only. All data and keys must be copied if you need to modify it.

Functions and structures

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\[linux-master][Documentation][admin-guide]bootconfig.rst, line 296)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/bootconfig.h
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\[linux-master][Documentation][admin-guide]bootconfig.rst, line 297)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: lib/bootconfig.c
```