Dynamic debug

Introduction

This document describes how to use the dynamic debug (dyndbg) feature.

Dynamic debug is designed to allow you to dynamically enable/disable kernel code to obtain additional kernel information. Currently, if `CONFIG_DYNAMIC_DEBUG` is set, then all `pr_debug()`/`dev_dbg()` and `print_hex_dump_debug()`/`print_hex_dump_bytes()` calls can be dynamically enabled per-callsite.

If you do not want to enable dynamic debug globally (i.e. in some embedded system), you may set `CONFIG_DYNAMIC_DEBUG_CORE` as basic support of dynamic debug and add `ccflags := -DDYNAMIC_DEBUG_MODULE` into the Makefile of any modules which you'd like to dynamically debug later.

If `CONFIG_DYNAMIC_DEBUG` is not set, `print_hex_dump_debug()` is just shortcut for `print_hex_dump(KERN_DEBUG)`.

For `print_hex_dump_debug()`/`print_hex_dump_bytes()`, format string is its `prefix_str` argument, if it is constant string; or `hexdump` in case `prefix_str` is built dynamically.

Dynamic debug has even more useful features:

- Simple query language allows turning on and off debugging statements by matching any combination of 0 or 1 of:
  - source filename
  - function name
  - line number (including ranges of line numbers)
  - module name
  - format string
- Provides a debugfs control file: `<debugfs>/dynamic_debug/control` which can be read to display the complete list of known debug statements, to help guide you

Controlling dynamic debug Behaviour

The behaviour of `pr_debug()`/`dev_dbg()` are controlled via writing to a control file in the 'debugfs' filesystem. Thus, you must first mount the debugfs filesystem, in order to make use of this feature. Subsequently, we refer to the control file as: `<debugfs>/dynamic_debug/control`. For example, if you want to enable printing from source file `svcsock.c`, line 1603 you simply do:

```
nullarbor:~ # echo 'file svcsock.c line 1603 +p' >
                            <debugfs>/dynamic_debug/control
```

If you make a mistake with the syntax, the write will fail thus:

```
nullarbor:~ # echo 'file svcsock.c wtf 1 +p' >
                            <debugfs>/dynamic_debug/control
-bash: echo: write error: Invalid argument
```

Note, for systems without 'debugfs' enabled, the control file can be found in `/proc/dynamic_debug/control`.

Viewing Dynamic Debug Behaviour

You can view the currently configured behaviour of all the debug statements via:

```
nullarbor:~ # cat <debugfs>/dynamic_debug/control
# filename:lineno [module]function flags format
net/sunrpc/svc_rdma.c:323 [svcxprt_rdma]svc_rdma_cleanup =_ "SVCRDMA Module Removed, deregister RPC RDMA transp
net/sunrpc/svc_rdma.c:341 [svcxprt_rdma]svc_rdma_init =_ "\011max_inline       : %d\012"
net/sunrpc/svc_rdma.c:340 [svcxprt_rdma]svc_rdma_init =_ "\011sq_depth         : %d\012"
net/sunrpc/svc_rdma.c:338 [svcxprt_rdma]svc_rdma_init =_ "\011max_requests     : %d\012"
...
```

You can also apply standard Unix text manipulation filters to this data, e.g.:

```
nullarbor:~ # grep -i rdma <debugfs>/dynamic_debug/control  | wc -l
62

nullarbor:~ # grep -i tcp <debugfs>/dynamic_debug/control | wc -l
42
```

The third column shows the currently enabled flags for each debug statement callsite (see below for definitions of the flags). The default value, with no flags enabled, is `=_`. So you can view all the debug statement callsites with any non-default flags:

```
nullarbor:~ # awk '$3 != "=_"' <debugfs>/dynamic_debug/control
# filename:lineno [module]function flags format
net/sunrpc/svcsock.c:1603 [sunrpc]svc_send p "svc_process: st_sendto returned %d\012"
```

Command Language Reference

At the lexical level, a command comprises a sequence of words separated by spaces or tabs. So these are all equivalent:

```
nullarbor:~ # echo -n 'file svcsock.c line 1603 +p' >
                            <debugfs>/dynamic_debug/control
nullarbor:~ # echo -n '  file   svcsock.c     line  1603 +p  ' >
                            <debugfs>/dynamic_debug/control
nullarbor:~ # echo -n 'file svcsock.c line 1603 +p' >
```

```
<debugfs>/dynamic_debug/control
```

Command submissions are bounded by a write() system call. Multiple commands can be written together, separated by `;` or `\n`:

```
~# echo "func pnpacpi_get_resources +p; func pnp_assign_mem +p" \
   > <debugfs>/dynamic_debug/control
```

If your query set is big, you can batch them too:

```
~# cat query-batch-file > <debugfs>/dynamic_debug/control
```

Another way is to use wildcards. The match rule supports `*` (matches zero or more characters) and `?` (matches exactly one character). For example, you can match all usb drivers:

```
~# echo "file drivers/usb/* +p" > <debugfs>/dynamic_debug/control
```

At the syntactical level, a command comprises a sequence of match specifications, followed by a flags change specification:

```
command ::= match-spec* flags-spec
```

The match-spec's are used to choose a subset of the known pr_debug() callsites to which to apply the flags-spec. Think of them as a query with implicit ANDs between each pair. Note that an empty list of match-specs will select all debug statement callsites.

A match specification comprises a keyword, which controls the attribute of the callsite to be compared, and a value to compare against. Possible keywords are::

```
match-spec ::= 'func' string |
               'file' string |
               'module' string |
               'format' string |
               'line' line-range

line-range ::= lineno |
               '-'lineno |
               lineno'-' |
               lineno'-'lineno

lineno ::= unsigned-int
```

> **Note**
>
> `line-range` cannot contain space, e.g. "1-30" is valid range but "1 - 30" is not.

The meanings of each keyword are:

func

The given string is compared against the function name of each callsite. Example:

```
func svc_tcp_accept
func *recv*             # in rfcomm, bluetooth, ping, tcp
```

file

The given string is compared against either the src-root relative pathname, or the basename of the source file of each callsite. Examples:

```
file svcsock.c
file kernel/freezer.c   # ie column 1 of control file
file drivers/usb/*      # all callsites under it
file inode.c:start_*    # parse :tail as a func (above)
file inode.c:1-100      # parse :tail as a line-range (above)
```

module

The given string is compared against the module name of each callsite. The module name is the string as seen in `lsmod`, i.e. without the directory or the `.ko` suffix and with `-` changed to `_`. Examples:

```
module sunrpc
module nfsd
module drm*     # both drm, drm_kms_helper
```

format

The given string is searched for in the dynamic debug format string. Note that the string does not need to match the entire format, only some part. Whitespace and other special characters can be escaped using C octal character escape `\ooo` notation, e.g. the space character is `\040`. Alternatively, the string can be enclosed in double quote characters (`"`) or single quote characters (`'`). Examples:

```
format svcrdma:         // many of the NFS/RDMA server pr_debugs
format readahead        // some pr_debugs in the readahead cache
format nfsd:\040SETATTR // one way to match a format with whitespace
format "nfsd: SETATTR"  // a neater way to match a format with whitespace
format 'nfsd: SETATTR'  // yet another way to match a format with whitespace
```

line

The given line number or range of line numbers is compared against the line number of each `pr_debug()` callsite. A single line number matches the callsite line number exactly. A range of line numbers matches any callsite between the first and last line number inclusive. An empty first number means the first line in the file, an empty last line number means the last line number in the file. Examples:

```
line 1603           // exactly line 1603
line 1600-1605      // the six lines from line 1600 to line 1605
line -1605          // the 1605 lines from line 1 to line 1605
line 1600-          // all lines from line 1600 to the end of the file
```

The flags specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

```
-    remove the given flags
+    add the given flags
=    set the flags to the given flags
```

The flags are:

```
p    enables the pr_debug() callsite.
f    Include the function name in the printed message
l    Include line number in the printed message
m    Include module name in the printed message
t    Include thread ID in messages not generated from interrupt context
_    No flags are set. (Or'd with others on input)
```

For `print_hex_dump_debug()` and `print_hex_dump_bytes()`, only `p` flag have meaning, other flags ignored.

For display, the flags are preceded by `=` (mnemonic: what the flags are currently equal to).

Note the regexp `^[-+=][flmpt_]+$` matches a flags specification. To clear all flags at once, use `=_` or `-flmpt`.

Debug messages during Boot Process

To activate debug messages for core code and built-in modules during the boot process, even before userspace and debugfs exists, use `dyndbg="QUERY"` or `module.dyndbg="QUERY"`. QUERY follows the syntax described above, but must not exceed 1023 characters. Your bootloader may impose lower limits.

These `dyndbg` params are processed just after the ddebug tables are processed, as part of the early_initcall. Thus you can enable debug messages in all code run after this early_initcall via this boot parameter.

On an x86 system for example ACPI enablement is a subsys_initcall and:

```
dyndbg="file ec.c +p"
```

will show early Embedded Controller transactions during ACPI setup if your machine (typically a laptop) has an Embedded Controller. PCI (or other devices) initialization also is a hot candidate for using this boot parameter for debugging purposes.

If `foo` module is not built-in, `foo.dyndbg` will still be processed at boot time, without effect, but will be reprocessed when module is loaded later. Bare `dyndbg=` is only processed at boot.

Debug Messages at Module Initialization Time

When `modprobe foo` is called, modprobe scans `/proc/cmdline` for `foo.params`, strips `foo.`, and passes them to the kernel along with params given in modprobe args or `/etc/modprob.d/*.conf` files, in the following order:

1. parameters given via `/etc/modprobe.d/*.conf`:

   ```
   options foo dyndbg=+pt
   options foo dyndbg # defaults to +p
   ```

2. `foo.dyndbg` as given in boot args, `foo.` is stripped and passed:

   ```
   foo.dyndbg=" func bar +p; func buz +mp"
   ```

3. args to modprobe:

   ```
   modprobe foo dyndbg==pmf # override previous settings
   ```

These `dyndbg` queries are applied in order, with last having final say. This allows boot args to override or modify those from `/etc/modprobe.d` (sensible, since 1 is system wide, 2 is kernel or boot specific), and modprobe args to override both.

In the `foo.dyndbg="QUERY"` form, the query must exclude `module foo`. `foo` is extracted from the param-name, and applied to each query in `QUERY`, and only 1 match-spec of each type is allowed.

The `dyndbg` option is a "fake" module parameter, which means:

- modules do not need to define it explicitly
- every module gets it tacitly, whether they use pr_debug or not
- it doesn't appear in `/sys/module/$module/parameters/` To see it, grep the control file, or inspect `/proc/cmdline`.

For `CONFIG_DYNAMIC_DEBUG` kernels, any settings given at boot-time (or enabled by `-DDEBUG` flag during compilation) can be disabled later via the debugfs interface if the debug messages are no longer needed:

```
echo "module module_name -p" > <debugfs>/dynamic_debug/control
```

Examples

```
// enable the message at line 1603 of file svcsock.c
nullarbor:~ # echo -n 'file svcsock.c line 1603 +p' >
                            <debugfs>/dynamic_debug/control

// enable all the messages in file svcsock.c
nullarbor:~ # echo -n 'file svcsock.c +p' >
                            <debugfs>/dynamic_debug/control
```

```
// enable all the messages in the NFS server module
nullarbor:~ # echo -n 'module nfsd +p' >
                              <debugfs>/dynamic_debug/control

// enable all 12 messages in the function svc_process()
nullarbor:~ # echo -n 'func svc_process +p' >
                              <debugfs>/dynamic_debug/control

// disable all 12 messages in the function svc_process()
nullarbor:~ # echo -n 'func svc_process -p' >
                              <debugfs>/dynamic_debug/control

// enable messages for NFS calls READ, READLINK, READDIR and READDIR+.
nullarbor:~ # echo -n 'format "nfsd: READ" +p' >
                              <debugfs>/dynamic_debug/control

// enable messages in files of which the paths include string "usb"
nullarbor:~ # echo -n 'file *usb* +p' > <debugfs>/dynamic_debug/control

// enable all messages
nullarbor:~ # echo -n '+p' > <debugfs>/dynamic_debug/control

// add module, function to all enabled messages
nullarbor:~ # echo -n '+mf' > <debugfs>/dynamic_debug/control

// boot-args example, with newlines and comments for readability
Kernel command line: ...
  // see whats going on in dyndbg=value processing
  dynamic_debug.verbose=3
  // enable pr_debugs in the btrfs module (can be builtin or loadable)
  btrfs.dyndbg="+p"
  // enable pr_debugs in all files under init/
  // and the function parse_one, #cmt is stripped
  dyndbg="file init/* +p #cmt ; func parse_one +p"
  // enable pr_debugs in 2 functions in a module loaded later
  pc87360.dyndbg="func pc87360_init_device +p; func pc87360_find +p"
```