NAME

DEBUG_FP, **KFAIL_POINT_CODE**, **KFAIL_POINT_CODE_FLAGS**, **KFAIL_POINT_CODE_COND**, **KFAIL_POINT_ERROR**, **KFAIL_POINT_EVAL**, **KFAIL_POINT_DECLARE**, **KFAIL_POINT_DEFINE**, **KFAIL_POINT_GOTO**, **KFAIL_POINT_RETURN**, **KFAIL_POINT_RETURN_VOID**, **KFAIL_POINT_SLEEP_CALLBACKS**, **fail_point -** fail points

SYNOPSIS

#include <sys/fail.h>

KFAIL_POINT_CODE(*parent*, *name*, *code*);

KFAIL_POINT_CODE_FLAGS(*parent*, *name*, *flags*, *code*);

KFAIL_POINT_CODE_COND(*parent*, *name*, *cond*, *flags*, *code*);

KFAIL_POINT_ERROR(*parent*, *name*, *error_var*);

KFAIL_POINT_EVAL(*name*, *code*);

KFAIL_POINT_DECLARE(*name*);

KFAIL_POINT_DEFINE(*parent*, *name*, *flags*);

KFAIL_POINT_GOTO(*parent*, *name*, *error_var*, *label*);

KFAIL_POINT_RETURN(*parent*, *name*);

KFAIL_POINT_RETURN_VOID(*parent*, *name*);

KFAIL_POINT_SLEEP_CALLBACKS(*parent*, *name*, *pre_func*, *pre_arg*, *post_func*, *post_arg*, *code*);

DESCRIPTION

Fail points are used to add code points where errors may be injected in a user controlled fashion. Fail points provide a convenient wrapper around user-provided error injection code, providing a sysctl(9) MIB, and a parser for that MIB that describes how the error injection code should fire.

The base fail point macro is **KFAIL_POINT_CODE**() where *parent* is a sysctl tree (frequently **DEBUG_FP** for kernel fail points, but various subsystems may wish to provide their own fail point trees), and *name* is the name of the MIB in that tree, and *code* is the error injection code. The *code* argument does not require braces, but it is considered good style to use braces for any multi-line code arguments. Inside the *code* argument, the evaluation of **RETURN_VALUE** is derived from the **return**() value set in the sysctl MIB.

Additionally, **KFAIL_POINT_CODE_FLAGS**() provides a *flags* argument which controls the fail point’s behaviour. This can be used to e.g., mark the fail point’s context as non-sleepable, which causes the **sleep** action to be coerced to a busy wait. The supported flags are:

FAIL_POINT_USE_TIMEOUT_PATH
Rather than sleeping on a **sleep**() call, just fire the post-sleep function after a timeout fires.

FAIL_POINT_NONSLEEPABLE
Mark the fail point as being in a non-sleepable context, which coerces **sleep**() calls to **delay**() calls.

Likewise, **KFAIL_POINT_CODE_COND**() supplies a *cond* argument, which allows you to set the condition under which the fail point’s code may fire. This is equivalent to:

```
if (cond)
        KFAIL_POINT_CODE_FLAGS(...);
```

See *SYSCTL VARIABLES* below.

The remaining **KFAIL_POINT_*()** macros are wrappers around common error injection paths:

KFAIL_POINT_RETURN(*parent*, *name*) is the equivalent of **KFAIL_POINT_CODE(..., return RETURN_VALUE)**

KFAIL_POINT_RETURN_VOID(*parent*, *name*) is the equivalent of **KFAIL_POINT_CODE(..., return)**

KFAIL_POINT_ERROR(*parent*, *name*, *error_var*) is the equivalent of **KFAIL_POINT_CODE(..., error_var = RETURN_VALUE)**

KFAIL_POINT_GOTO(*parent*, *name*, *error_var*, *label*) is the equivalent of **KFAIL_POINT_CODE(..., { error_var = RETURN_VALUE; goto label;})**

You can also introduce fail points by separating the declaration, definition, and evaluation portions.

KFAIL_POINT_DECLARE(*name*) is used to declare the **fail_point** struct.

KFAIL_POINT_DEFINE(*parent*, *name*, *flags*) defines and initializes the **fail_point** and sets up its sysctl(9).

KFAIL_POINT_EVAL(*name*, *code*) is used at the point that the fail point is executed.

SYSCTL VARIABLES

The **KFAIL_POINT_*()** macros add sysctl MIBs where specified. Many base kernel MIBs can be found in the **debug.fail_point** tree (referenced in code by **DEBUG_FP**).

The sysctl variable may be set in a number of ways:

```
  [<pct>%][<cnt>*]<type>[(args...)][-><more terms>]
```

The <type> argument specifies which action to take; it can be one of:

off Take no action (does not trigger fail point code)

return Trigger fail point code with specified argument

sleep Sleep the specified number of milliseconds

panic Panic

break Break into the debugger, or trap if there is no debugger support

print Print that the fail point executed

pause Threads sleep at the fail point until the fail point is set to **off**

yield Thread yields the cpu when the fail point is evaluated

delay Similar to sleep, but busy waits the cpu. (Useful in non-sleepable contexts.)

The <pct>% and <cnt>* modifiers prior to <type> control when <type> is executed. The <pct>% form (e.g. "1.2%") can be used to specify a probability that <type> will execute. This is a decimal in the range (0, 100] which can specify up to 1/10,000% precision. The <cnt>* form (e.g. "5*") can be used to specify the number of times <type> should be executed before this <term> is disabled. Only the last probability and the last count are used if multiple are specified, i.e. "1.2%2%" is the same as "2%". When both a probability and a count are specified, the probability is evaluated before the count, i.e. "2%5*" means "2% of the time, but only 5 times total".

The operator -> can be used to express cascading terms. If you specify <term1>-><term2>, it means that if <term1> does not ‘execute’, <term2> is evaluated. For the purpose of this operator, the **return**() and **print**() operators are the only types that cascade. A **return**() term only cascades if the code executes, and a **print**() term only cascades when passed a non-zero argument. A pid can optionally be specified. The fail point term is only executed when invoked by a process with a matching p_pid.

EXAMPLES

sysctl debug.fail_point.foobar="2.1%return(5)"

21/1000ths of the time, execute *code* with RETURN_VALUE set to 5.

sysctl debug.fail_point.foobar="2%return(5)->5%return(22)"

2/100ths of the time, execute *code* with RETURN_VALUE set to 5. If that does not happen, 5% of the time execute *code* with RETURN_VALUE set to 22.

sysctl debug.fail_point.foobar="5*return(5)->0.1%return(22)"

For 5 times, return 5. After that, 1/1000th of the time, return 22.

sysctl debug.fail_point.foobar="0.1%5*return(5)"

Return 5 for 1 in 1000 executions, but only 5 times total.

sysctl debug.fail_point.foobar="1%*sleep(50)"

1/100th of the time, sleep 50ms.

sysctl debug.fail_point.foobar="1*return(5)[pid 1234]"

Return 5 once, when pid 1234 executes the fail point.

AUTHORS

This manual page was written by

Matthew Bryan <*matthew.bryan@isilon.com*> and

Zach Loafman <*zml@FreeBSD.org*>.

CAVEATS

It is easy to shoot yourself in the foot by setting fail points too aggressively or setting too many in combination. For example, forcing **malloc**() to fail consistently is potentially harmful to uptime.

The **sleep**() sysctl setting may not be appropriate in all situations. Currently, **fail_point_eval**() does not verify whether the context is appropriate for calling **msleep**(). You can force it to evaluate a **sleep** action as a **delay** action by specifying the **FAIL_POINT_NONSLEEPABLE** flag at the point the fail point is declared.