NAME

stack - kernel thread stack tracing routines

SYNOPSIS

#include <sys/param.h>
#include <sys/stack.h>

In the kernel configuration file:
options DDB
options STACK

*struct stack **
stack_create(*int flags*);

void
stack_destroy(*struct stack *st*);

int
stack_put(*struct stack *st, vm_offset_t pc*);

void
stack_copy(*const struct stack *src, struct stack *dst*);

void
stack_zero(*struct stack *st*);

void
stack_print(*const struct stack *st*);

void
stack_print_ddb(*const struct stack *st*);

void
stack_print_short(*const struct stack *st*);

void
stack_print_short_ddb(*const struct stack *st*);

void
stack_sbuf_print(*struct sbuf *sb, const struct stack *st*);

void
stack_sbuf_print_ddb(*struct sbuf *sb, const struct stack *st*);

void
stack_save(*struct stack *st*);

int
stack_save_td(*struct stack *st, struct thread *td*);

DESCRIPTION

The **stack** KPI allows querying of kernel stack trace information and the automated generation of kernel stack trace strings for the purposes of debugging and tracing. To use the KPI, at least one of **options DDB** and **options STACK** must be compiled into the kernel.

Each stack trace is described by a *struct stack*. It can be declared in the usual ways, including on the stack, and optionally initialized with **stack_zero**(), though this is not necessary before saving a trace. It can also be dynamically allocated with **stack_create**(). The *flags* argument is passed to malloc(9). This dynamic allocation must be freed with **stack_destroy**().

A trace of the current thread's kernel call stack may be captured using **stack_save**(). **stack_save_td**() can be used to capture the kernel stack of a caller-specified thread. Callers of **stack_save_td**() must own the thread lock of the specified thread, and the thread's stack must not be swapped out. **stack_save_td**() can capture the kernel stack of a running thread, though note that this is not implemented on all platforms. If the thread is running, the caller must also hold the process lock for the target thread.

stack_print() and **stack_print_short**() may be used to print a stack trace using the kernel printf(9), and may sleep as a result of acquiring sx(9) locks in the kernel linker while looking up symbol names. In locking-sensitive environments, the unsynchronized **stack_print_ddb**() and **stack_print_short_ddb**() variants may be invoked. This function bypasses kernel linker locking, making it usable in ddb(4), but not in a live system where linker data structures may change.

stack_sbuf_print() may be used to construct a human-readable string, including conversion (where possible) from a simple kernel instruction pointer to a named symbol and offset. The argument *sb* must be an initialized struct sbuf as described in sbuf(9). This function may sleep if an auto-extending struct sbuf is used, or due to kernel linker locking. In locking-sensitive environments, such as ddb(4), the unsynchronized **stack_sbuf_print_ddb**() variant may be invoked to avoid kernel linker locking; it should be used with a fixed-length sbuf.

The utility functions **stack_zero**, **stack_copy**, and **stack_put** may be used to manipulate stack data structures directly.

RETURN VALUES

stack_put() returns 0 on success. Otherwise the struct stack does not contain space to record additional frames, and a non-zero value is returned.

stack_save_td() returns 0 when the stack capture was successful and a non-zero error number otherwise. In particular, EBUSY is returned if the thread was running in user mode at the time that the capture was attempted, and EOPNOTSUPP is returned if the operation is not implemented.

SEE ALSO

ddb(4), printf(9), sbuf(9), sx(9)

AUTHORS

The **stack** function suite was created by Antoine Brodin. **stack** was extended by Robert Watson for general-purpose use outside of ddb(4).